

A few words of welcome from the General Chair

It is a great pleasure to have the SIGOPS EW in France again, 10 years after the Mont Saint-Michel workshop. While Mont Saint-Michel is surrounded by the sea, Saint-Emilion is amidst miles of vineyards. There is no tide here, even if one may refer to the level of wine in the bottles... Still, I am sure that Emilion will do as well as Michel did in inspiring fruitful discussions among us. In the event of a failure of the saint's spirit, I am sure that the wine cellar located underneath the meeting room in "Le chateau la Couspaude", a *Saint-Emilion grand cru classé*, will be an excellent backup.

"Can we really depend on an OS?", "In vino veritas", Baccus would have certainly answered.

Thank you for coming to EW2002 and have a very enriching workshop,

Gilles Muller
EW2002 General Chair
Nantes, July 2002

A few words of welcome from the Program Chair

First, thanks to all the authors for their essential efforts to make this workshop successful; without good papers—no workshop. We had a total of 50 submissions which is slightly up from the previous workshop. Of these the PC selected sixteen for presentations; thanks to the PC for their work.

I used a simple system for submission: merely e-mail. This, unfortunately, does not work perfectly; I "lost" several papers which delayed their review—apologies to those authors that had to wait past the notification deadline.

The European Workshop is usually held in beautiful locations where the relaxed atmosphere can foster fruitful discussions between the participants. Gilles Muller has found a pearl of a location—and not just for wine lovers! And he has put in a tremendous effort to do the arrangements. I hope that you will all participate actively.

The theme of the workshop "Can we really depend on an OS?" has been viewed from many different points as the presentations will show. The panel discussion will attack the issues head-on.

Finally, thanks goes to Jean-Marc Menaud for his help with registration and to Robert Bialek and Christian Boesgaard for their efforts in producing the proceedings.

Have fun!

Eric Jul
EW2002 Program Chair
Copenhagen, July 2002

Contents

1	List of Papers	3
2	List of extended abstracts	4
3	List of authors	6
4	Session Papers	9
	Session 1. Instrumentation	9
	Session 2. Operating Systems	30
	Session 3. Theory	44
	Session 4. Robust Service	62
	Session 5. Security & Authentication	78
	Session 6. Operating System's Structures	101
	Session 7. Peer-to-Peer	116
5	Extended Abstracts	146

List of Papers

Title	Author	page
Brittle Systems will Break Not Bend: Can Aspect-Oriented Programming Help?	Yvonne Coady, Gregor Kiczales	78
Capturing OS Expertise in a Modular Type System: the Bossa Experience	Julia L. Lawall, Gilles Muller, Luciano Porto Barreto	54
Denali: A Scalable Isolation Kernel	Andrew Whitaker, Marianne Shaw, Steven D. Gribble	9
Dependable software needs pervasive debugging	Timothy L. Harris	38
HiScamp: self-organizing hierarchical membership protocol	Ayalvadi J. Ganesh, Anne-Marie Kermarrec, Laurent Massoulie	133
InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems	Timothy Roscoe, Richard Mortier, Paul Jardetzky, Steven Hand	30
Nooks: An Architecture for Reliable Device Drivers	Michael M. Swift, Steven Martin, Henry M. Levy, Susan G. Eggers	101
One Ring To Rule Them All: Service discovery and binding with a distributed hash table	Miguel Castro, Peter Druschel, Antony Rowstron	140
Overload Management as a Fundamental Service Design Primitive	Matt Welsh, David Culler	62
Rewind, Repair, Replay: Three R's to Dependability	Aaron B. Brown, David A. Patterson	70
Rigour is good for you and feasible: reflection on formal treatments of C and UDP sockets	Michael Norrish, Peter Sewell, Keith Wansbrough	49
Security Architectures Revisited	Hermann Hartig	16
Self-Organization in Peer-to-Peer Systems	Jonathan Ledlie, Jacob Taylor, Laura Serban, Margo Seltzer	125
Specifying and Verifying Systems With TLA+	Leslie Lamport, John Matthews, Mark Tuttle, Yuan Yu	44
Sub-Operating Systems: A New Approach to Application Security	Sotiris Ioannidis, Steven M. Bellovin, Jonathan M. Smith	108
The Case for Cyber Foraging	M. Satyanarayanan, Rejesh Balan, Shafeeq Sinnamohideen, Jason Flinn, Hen-I Yang	87
The Case for Transient Authentication	Brian D. Noble, Mark D. Corner	24
The Design of a Robust Peer-to-Peer System	Rodrigo Rodrigues, Barbara Liskov, Liuba Shrira	116

List of extended abstracts

Title	Author	page
A Design of the Persistent Operating System with Non-volatile Memory	Ren Ohmura, Nobuyuki Yamasaki, Yuichiro Anzai	148
AIMS: Robustness Through Sensible Introspection	Febian E. Bustamante, Christian Poellabauer, Karsten Schwan	153
An Approach for a Dependable Java Embedded Environment	Gilbert Caballic, Salam Majoul, Jean-Philippe Lesot, Michel Banatre	157
An Online Evolutionary Approach to Developing Internet Services	Mike Y. Chen, Emre Kiciman, Eric Brewer	161
Applying source-code verification to a microkernel - The Viasco project	Michael Howmuth, Hendrik Tews, Shane G. Stephens	165
Back to the Future: dependable computing = dependable services	Jeffrey Chase, Amin Vahdat, John Wilkes	170
Dependency on O.S. In long-term programs: Experience report in space programs	Patrick Cormery, Le Vinh Quy Ribal, Arnaud Stransky	174
Design and Implementation of the Lambda u-Kernel based Operating System for Embedded Systems	Kenji Hisazumi, Teruaki Kitasuka, Tsuneo Nakanishi, Akira Fukuda	178
Efficient Heartbeats and Repair of Softstate in Distributed Hash Table Systems	Hakim Weatherspoon and John D. Kubiatowicz	182
Event-driven Programming for Robust software	Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazieres, Robert Morris	186
Execution Time Limitation of Interrupt Handlers in a Java Operating System	Meik Felser, Michael Golm, Christian Wawersich, Juergen Kleinoder	190
Extensible distributed Operating System for reliable control systems	Katsumi Maruyama, Kazuya Kadama, Soichiro Hidaka, Hiro-tatsu Hashizume	194
Fault Tolerance in Biomedical Systems	Shane Stephens	198
Gaining and Maintaining Confidence in Operating Systems Security	Trent Jaeger, Antony Edwards, Xiaolan Zhang	201
High-Confidence Operatins Systems	Radu Grosu, Erez Zadok, Scott A. Smolka, Rance Cleaveland, and Yanhong A. Liu	205
Increasing smart card dependability	Ludovic Casset, Jean-Louis Lanet	209
Making Sound Tradeoffs in State Management	George Candea, Armando Fox	213
Model Checking System Software with CMC	Madanlal Musuvathi, Andy Chou, David Dill, Dawson Engler	219
OASIS project: deterministic real-time for safety critical embedded systems	Stephane Louise, Vincent David, Jean Delcoigne, Christophe Ausagues	223

Operating System Support for Massive Replication	Arun Venkataramani, Ravindranath Kokku, Mike Dahlin	227
Pangaea: a symbolic wide-area file system	Yasushi Saito, Christos Karamanolis	231
Replica Management Should Be A Game	Dennis Geels, John Kubiawicz	235
Secure Coprocessor-based Intrusion Detection	Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ron Perez, Reiner Sailer	239
THINK: a secure distributed systems architecture	Christophe Rippert, Jean-Bernards Stefani	243
Time Fault Removal for Safety-Critical Real-Time Embedded Systems	Sebastien Faucou, Anne-Marie Depanche, Yvon Trinquet	247
Towards Trusted Systems from the Ground Up	Vivek Haldar, Michael Franz	251
Why do internet services fail and what can be done about it?	David Oppenheimer, David A. Patterson	255

List of authors

Author	e-mail	Title	page
Aaron B. Brown	abrown@cs.berkeley.edu	Rewind, Repair, Replay. Three R's to Dependability	70
Andrew Whitaker	andrew@cs.washington.edu	Denali: A Scalable Isolation Kernel	9
Anne-Marie Kermarrec	annemk@microsoft.com	HiScamp: self-organizing hierarchical membership protocol	133
Antony Rowstron	antr@microsoft.com	One Ring To Rule Them All: Service discovery and binding with a distributed hash table	140
Ayalvadi J. Ganesh	ajg@microsoft.com	HiScamp: self-organizing hierarchical membership protocol	133
Barbara Liskov	liskov@lcs.mit.edu	The Design of a Robust Peer-to-Peer System	116
Brian D. Noble	bnoble@umich.edu	The Case for Transient Authentication	24
David Culler	culler@cs.berkeley.edu	Overload Management as a Fundamental Service Design Primitive	62
David A. Patterson		Rewind, Repair, Replay: Three s to Dependability	70
Gilles Muller	Gilles.muller@inria.fr	Capturing OS Expertise in a Modular Type System: the Bossa Experience	54
Gregor Kiczales		Brittle Systems will Break Not Bend: Can Aspect-Oriented Programming Help?	78
Hen-I Yang		The Case for Cyber Foraging	87
Henry M. Levy	levy@cs.washington.edu	Nooks: An Architecture for Reliable Device Drivers	101
Hermann Hartig	haertig@os.inf.tu-dresden.de	Security Architectures Revisited	16
Jacob Taylor	jonathan@eecs.harvard.edu	Self-Organization in Peer-to-Peer Systems	125
Jason Flinn		The Case for Cyber Foraging	87
John Matthews		Specifying and Verifying Systems With TLA+	44
John Wilkes	wilkes@hpl.hp.com	Automatic data dependability	93
Jonathan Ledlie	jonathan@eecs.harvard.edu	Self-Organization in Peer-to-Peer Systems	125
Jonathan M. Smith	jms@dsl.cis.upenn.edu	Sub-Operating Systems: A New Approach to Application Security	108
Julia L. Lawall	julia@diku.dk	Capturing OS Expertise in a Modular Type System: the Bossa Experience	54

Keith Wansbrough	Keith.wansbrough@cl.cam.ac.uk	Rigour is good for you and feasible: reflection on formal treatments of C and UDP sockets	49
Kimberly Keeton	kkeeton@hpl.hp.com	Automatic data dependability	93
Laura Serban	serban@fas.harvard.edu	Self-Organization in Peer-to-Peer Systems	125
Laurent Massoulie	lmassoul@microsoft.com	HiScamp: self-organizing hierarchical membership protocol	133
Leslie Lamport	lamport@microsoft.com	Specifying and Verifying Systems With TLA+	44
Liuba Shrira	liuba@lcs.mit.edu	The Design of a Robust Peer-to-Peer System	116
Luciano Porto Barreto	barreto@labri.fr	Capturing OS Expertise in a Modular Type System: the Bossa Experience	54
M. Satyanarayanan		The Case for Cyber Foraging	87
Margo Seltzer	margo@eecs.harvard.edu	Self-Organization in Peer-to-Peer Systems	125
Marianne Shaw	mar@cs.washington.edu	Denali: A Scalable Isolation Kernel	9
Mark D. Corner		The Case for Transient Authentication	24
Mark Tuttle		Specifying and Verifying Systems With TLA+	44
Matt Welsh	mdw@cs.berkeley.edu	Overload Management as a Fundamental Service Design Primitive	62
Michael M. Swift	mikesw@cs.washington.edu	Nooks: An Architecture for Reliable Device Drivers	101
Michael Norrish	Michael.norrish@cl.cam.ac.uk	Rigour is good for you and feasible: reflection on formal treatments of C and UDP sockets	49
Miguel Castro	mcastro@microsoft.com	One Ring To Rule Them All: Service discovery and binding with a distributed hash table	140
Paul Jardetzky		InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems	30
Peter Druschel	druschel@cs.rice.edu	One Ring To Rule Them All: Service discovery and binding with a distributed hash table	140
Peter Sewell	peter.sewell@cl.cam.ac.uk	Rigour is good for you and feasible: reflection on formal treatments of C and UDP sockets	49
Rejesh Balan	rajesh@cs.cmu.edu	The Case for Cyber Foraging	87
Richard Mortier		InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems	30
Rodrigo Rodrigues	rodrigo@lcs.mit.edu	The Design of a Robust Peer-to-Peer System	116

Shafeeq Sinnamohideen		The Case for Cyber Foraging	87
Sotiris Ioannidis	sotiris@dsl.cis.upenn.edu	Sub-Operating Systems: A New Approach to Application Security	108
Steven D. Gribble	gribble@cs.washington.edu	Denali: A Scalable Isolation Kernel	9
Steven Hand		InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems	30
Steven M. Bellovin	smb@research.att.com	Sub-Operating Systems: A New Approach to Application Security	108
Steven Martin	stevaroo@cs.washington.edu	Nooks: An Architecture for Reliable Device Drivers	101
Susan G. Eggers	eggers@cs.washington.edu	Nooks: An Architecture for Reliable Device Drivers	101
Timothy L. Harris	tim.harris@cl.cam.ac.uk	Dependable software needs pervasive debugging	38
Timothy Roscoe	troscoe@intel-research.net	InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems	30
Yuan Yu		Specifying and Verifying Systems With TLA+	44
Yvonne Coady	ycoady@cs.ubc.ca	Brittle Systems will Break Not Bend: Can Aspect-Oriented Programming Help?	78

Session Papers

Session 1. Instrumentation

1. Denali: A Scalable Isolation Kernel
Authors: Andrew Whitaker, Marianne Shaw, Steven D. Gribble
page 9
2. Security Architectures Revisited
Authors: Hermann Hartig
page 16
3. The Case for Transient Authentication
Authors: Brian D. Noble, Mark D. Corner
page 24

Denali: A Scalable Isolation Kernel

Andrew Whitaker, Marianne Shaw, and Steven D. Gribble

The University of Washington

{andrew,mar,gribble}@cs.washington.edu

Abstract

The Denali project provides system support for running several mutually distrusting Internet services on the same physical infrastructure. For example, this would enable a developer to push dynamic content into third party hosting infrastructure such as content distribution networks. To accomplish this, we propose a new kernel architecture called an isolation kernel to isolate untrusted applications. An isolation kernel is a simple, thin software layer that runs directly on hardware (and hence below operating systems), whose function is to subdivide a physical machine into a set of fully isolated protection domains. Isolation kernels resemble virtual machine monitors in that they expose a virtualized hardware interface to a set of virtual machines. Unlike VMMs, however, isolation kernels do not attempt to precisely emulate the underlying physical architecture. By selectively modifying the hardware architecture, we enable our system to scale up to 1000's of virtual machines on commodity hardware. In this paper, we describe a set of design principles that govern isolation kernels, briefly discuss a prototype isolation kernel, and present future work and applications of isolation kernels.

1. Introduction

The Internet is dramatically changing the way we think about application deployment. Instead of installing shrink-wrapped software on their PCs, users are increasingly relying on infrastructural services such as Hotmail and MapQuest. This “Internet services” model offers several powerful advantages: software distribution is simplified, upgrades and bug-fixes can be applied immediately, and services are always on and accessible from any capable device. The attractiveness of this model has resulted in significant industrial and research attention into Internet service frameworks, including .NET.

Although Internet services have compelling advantages, the introduction of a new service currently requires a large investment in infrastructure and administration. We believe this high barrier to entry stifles innovation, especially when the service must run in many locations across the wide-area (as would be necessary to enable dynamically generated content in content delivery networks, for example). For the Internet services model to succeed, we believe that the ownership and management of physical Internet service infrastructure is best handled by third party providers such as

ISPs, and that mechanisms should be established to allow Internet service authors to “push” new services into this infrastructure in a safe manner.

Because not all services warrant their own dedicated hardware, services will need to be multiplexed on the same physical machines, as is currently done for virtual web site management. Unfortunately, Internet services contain active code rather than static data, which raises serious trust and security issues: infrastructure providers cannot trust hosted Internet services, and services will not trust each other. Accordingly, there is a need to provide strong isolation between services, both to enforce security and to control services’ resource consumption.

In this paper, we propose a software construct called an *isolation kernel*, whose purpose is to multiplex physical hardware across many mutually untrusting Internet services by containing each service within an isolation domain. Although there have been attempts to address security and performance isolation within a monolithic OS [2, 10], we believe these issues are best addressed *under* a monolithic OS. An isolation kernel is a thin software layer which virtualizes the underlying hardware, in much the same fashion as a virtual machine monitor (VMM). Unlike a VMM, an isolation kernel does not attempt to precisely emulate the underlying physical architecture, because there are significant simplicity, scalability, and performance benefits to be gained by modifying it, as we will argue later in this paper.

An isolation kernel is similar in many respects to other small-kernel architectures, such as virtual machine monitors [12], hypervisors [4], microkernels [1], and exokernels [8]. In the next section of this paper, we outline some of the guiding design principles of isolation kernels, and describe how our work differs from other small-kernel architectures. Next, we describe the architecture and implementation of Denali, our prototype isolation kernel for the x86 architecture. Finally, we sketch out our future directions for our research.

2. Design Principles and Choices

In this section of the paper, we present a number of principles and design choices that guided us while architecting the Denali isolation kernel. These principles were motivated by technological trends, as well as characteristics of

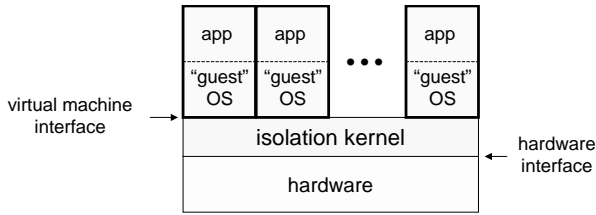


Figure 1. An isolation kernel. An isolation kernel is a thin software layer which exposes a virtual machine abstraction, and which prevents direct sharing across VMs.

the applications that we wish to support.

Simplicity promotes security: A long history of vulnerabilities suggests that conventional operating systems are poorly suited to isolating and containing untrusted code. We argue that this is a fundamental consequence of two architectural characteristics of OSs, rather than simply being a matter of avoidable implementation flaws.

First, an OS exposes high-level abstractions, rather than low-level resources, and enforces protection at the same layer as the exposed high-level abstractions. For example, files serve as an abstract container for persistent data, as well as a unit of access control. The high level at which protection is enforced gives rise to “layer-below attacks” [13], in which an attacker accesses a resource below the layer of abstraction. For example, forcing a core-dump of an application may allow an attacker to bypass virtual memory protection to inspect program state. The complexity of OSs, which is largely due to the complexity of implementing high-level abstractions, makes it extremely difficult to anticipate and protect against layer-below attacks.

Second, a modern OS typically exposes a wide API. The original version of UNIX had only 33 system calls; today, Windows has over 3400 system calls [17]. This wide API runs contrary to the principle of economy of mechanism [16], which states that security is achieved by limiting the number of access paths available to untrusted code. Monolithic OSs are constantly evolving: each new release contains millions of lines of new code and countless new features. This leads to insecurity since new code is known to contain more bugs than older, more stable code [6].

We believe that an isolation kernel must be implemented as a thin software layer that exposes a narrow API and runs directly on hardware; this is similar to small-kernel architectures such as virtual machine monitors or microkernels. Rather than providing complex high-level abstractions, an isolation kernel exposes and protects low-level hardware resources. This greatly simplifies the implementation of the isolation kernel, promoting security and avoiding layer-below attacks. Because an isolation kernel defers the implementation of abstractions to higher software layers (Figure 1), pressure to accumulate new features and widen its exposed interface is alleviated.

Performance is no longer the primary issue: In the past, small-kernel architectures have been considered inefficient when compared to monolithic kernels. However, technology advances have made raw performance less of an issue, and in many cases it has become reasonable to trade away performance in return for reliability, security, or manageability.

Additionally, recent results have demonstrated that the performance penalty of small kernels is not prohibitive. The Disco virtual machine monitor reported performance penalties of no more than 16% over a range of applications [5]. Our early experience with Denali is even more encouraging, as described in Section 3.

Sharing is infrequent: Operating systems have customarily provided efficient mechanisms for sharing data between applications, such as fast IPC and shared file systems. In our application domain, however, we expect sharing to be infrequent, since Internet services are designed and operated by independent users. This suggests that we can afford to have a high cost of sharing across isolation domains, if in return we strengthen isolation.

The emphasis of conventional OSs on providing data sharing mechanisms has a harmful effect on their ability to provide isolation. For example, in most OSs, all applications see the same global file system name space regardless of whether they want to share data. If the file system access control policy is not perfectly configured, then malicious applications can find hard-to-spot ways of reading or modifying other users’ data. Attackers have used symbolic links to trick unsuspecting applications into reading or modifying privileged system files¹. Although this particular vulnerability affects conventional OSs, we believe that small-kernel architectures that encourage fine-grained data sharing would be susceptible to the same class of vulnerabilities. This includes microkernel systems, which support sharing through user-mode file servers [1], and Exokernel systems which support sharing by downloading protection policy into the kernel [8].

Virtual machine monitors like Disco [5] and VM/370 are better suited to isolation because they disallow direct sharing. Each virtual machine on a VMM is confined to a private, virtual namespace: virtual physical memory pages, virtual disk blocks, and so forth. The default way to share data on such a system is to send the data over a (virtual) Ethernet segment. Thus, applications that desire complete isolation can simply ignore all network traffic, or use firewall techniques to filter out suspicious traffic.

Zipf’s law implies a need to scale: Measurement studies of web documents, web servers, DNS names, and other network services show that popularity distributions tend to be Zipfian [3]. Based on this, we expect that the popularity distribution for Internet services will also be driven by Zipf’s law.

Zipfian distributions are heavy-tailed, meaning that a

¹Refer to CERT vulnerability notes: VU#356323, VU#747736, and VU#426273.

non-trivial fraction of requests go to a large set of unpopular services. Individually, these services are accessed infrequently, motivating the desire to multiplex many of them on a single computer for reasons of affordability and manageability. However, the unpopular services collectively constitute a large fraction of the total requests. Previous studies of web cache performance have demonstrated that unpopular objects tend to drag down overall system performance [3].

Our goal in Denali is to support both popular and unpopular services. To support the unpopular, we want to be able to host a large number of services (hundreds, if not thousands) on a commodity PC. To make this feasible, we must use main memory as a cache of active services, using disk as backing store for the majority of services that are idle. Thus, our system must support rapid swapping of services off disk to mitigate the negative results in [3].

Transparency is a non-goal, and is potentially harmful: One consideration for small-kernel architectures is how to support legacy software, including OS code. Virtual machine monitors are distinct in that they can provide transparent backwards compatibility for legacy operating systems [5, 12] by precisely emulating the underlying physical architecture. Although transparency is useful for supporting legacy software, it is independent from the issue of enforcing isolation, and we believe there are compelling reasons to allow the interface exposed by an isolation kernel to differ from the physical architecture on which the kernel runs.

Some physical architectures are not strictly virtualizable [11]. Non-virtualizable architectures (such as x86) can be virtualized using binary re-writing techniques, however this requires significant complexity to handle a small set of infrequently used instructions [14]. Similarly, some components of the hardware or firmware are rarely used by applications, but must be emulated to maintain backwards compatibility with legacy OSs. Examples of this include the x86 segmentation hardware and the BIOS.

The Denali isolation kernel exposes an interface that is similar to the underlying hardware architecture, but with several strategic modifications. In the next section, we describe our architecture modifications, which we use to enhance scalability and performance, while simplifying the implementation of the isolation kernel and the services that run on it.

Of course, giving up backwards compatibility carries the disadvantage that we must modify legacy OS code to run on our architecture. On the other hand, breaking backwards compatibility has provided us with the opportunity to redesign the guest operating system to be virtualization-aware. In the following section, we describe how we have used this ability inside Denali.

3. Denali: A Scalable Isolation Kernel

The Denali kernel aims to provide strong isolation and high scalability for Internet services. To achieve isolation, the kernel exposes a virtualized hardware interface — vir-

tual I/O devices, virtual physical memory, etc. — to a set of virtual machines. Each virtual machine (VM) contains a guest OS, which contains a customary set of OS abstractions (such as TCP/IP sockets and threads), as well as one or more applications. In this respect, the Denali kernel resembles a virtual machine monitor.

Unlike VMMs, however, we have made selective modifications to the underlying physical architecture to promote scalability, performance and simplicity of implementation.

3.1. Architectural Modifications

One barrier to running a large number of concurrently active virtual machines is idle loops inside guest OSs. To prevent wasting CPU resources, Denali exposes an idle instruction, which allows a guest OS to relinquish control of the CPU². The VM idles until an external interrupt arrives that requires processing. To prevent a VM from idling forever, Denali exposes a virtual alarm clock, which is set by the guest OS to raise an interrupt after a given number of clock ticks.

Another barrier to scalability relates to the handling of timers. On most systems, the passage of time is indicated by a programmable interval timer, which generates an interrupt every few milliseconds. Emulating this behavior in an isolation kernel would require raising a virtual timer interrupt on each timer tick for each virtual machine, resulting in a large number of user/kernel crossings. Moreover, each clock tick would wake up idle virtual machines, thereby preventing unpopular services from being swapped out to disk.

Denali's approach is to only raise timer interrupts to the running virtual machine. For all other VMs, the kernel exposes a global clock that advances with the hardware clock. After a VM has been context switched, the kernel raises a "wakeup" interrupt to indicate that the guest OS should recalibrate timing-related routines against the global clock.

Denali's interrupt mechanism was designed to better support many concurrently executing VMs. As the number of VMs increases, it becomes likely that multiple virtual interrupts will arrive for a given VM while it is context-switched out. Denali delivers all pending interrupts to a virtual machine in a single batch rather than enforcing a strict serial ordering. Batching reduces the number of user/kernel crossings, and allows the system to piggyback "informational" interrupts on normal hardware interrupts.

The Denali architecture does not expose virtual memory management hardware; instead, virtual machines reside in a single, flat address space, much like conventional OS processes. As a result, guest OSs are directly linked against applications, similar to Exokernel library operating

²The Denali idle instruction is similar to the x86 hlt instruction, which places the processor in a halted state. However, the hlt instruction does not have a time limit, and therefore is only used after the processor has been idle for some period of time (usually hundreds of milliseconds). The Denali alarm clock mechanism allows for CPU sharing at a much finer granularity.

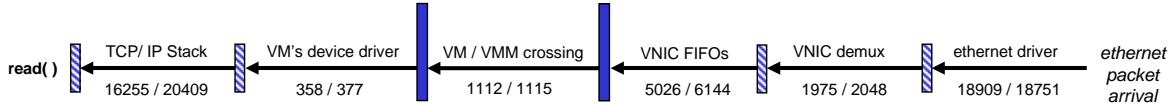


Figure 2. Packet processing overhead: This timeline illustrates the cost (in cycles) of packet reception, broken down across various functional stages. Each pair of numbers represents the number of cycles executed in that stage for 100 byte and 1400 byte packets, respectively. VNIC refers to the virtual NIC implementation.

systems [8]. We believe this simplification is justified, because a complex service that requires multiple protection domains can be structured to run in multiple virtual machines. Moreover, omitting virtual memory from the architecture improves performance by reducing TLB misses during guest OS context switches; this overhead was found to be significant during a recent measurement of VMWare’s workstation product [18].

Denali exposes virtual I/O devices to virtual machines, but the interfaces to these devices have been drastically simplified relative to real hardware devices. For example, the virtual Ethernet device supports two operations: packet send and packet receive. The interface to real hardware devices is often more complex than necessary, which can lead to reduced performance during virtualization [18].

The Denali architecture greatly simplifies virtual machine initialization. There is no BIOS exposed, and all Denali virtual devices “power on” in a well-known boot state, eliminating the need for a guest OS to initialize devices. These changes dramatically reduce the complexity of both our isolation kernel (since it doesn’t need to virtualize these architectural features), as well as the guest OSs themselves.

3.2. Implementation and Results

We have developed a prototype isolation kernel that runs directly on x86 hardware. Our implementation borrows device drivers and low-level support from the Flux OSKit [9]. However, the “core” of the kernel (scheduling, virtual device emulation, and paging) is entirely new. We have also constructed a prototype guest OS that contains a port of the BSD TCP/IP stack [7], full threading support, and a subset of the Posix API. We are currently investigating appropriate stable storage abstractions for our guest OS.

Our initial evaluation (on a 1.5 GHz Pentium IV uniprocessor with 1 GB RAM) has focused on networking applications, and has been very encouraging. A breakdown of the overhead associated with each network packet indicates that enforcing isolation through virtualization accounts for a relatively small fraction of the processing overhead (Figure 2). The physical device driver and guest OS TCP/IP stack represent the largest portion of per-packet overheads; for small and large packets, the physical device driver represents 43.3% and 38.4%, respectively, of the total receive overhead, while traversing of the TCP/IP stack accounted for an additional 37.3% and 41.8%, respectively.

Our current prototype can support an almost arbitrary

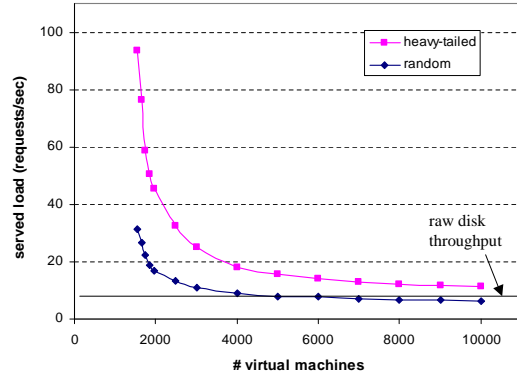


Figure 3. Web server throughput: This graph shows the aggregate serve load sustained across the set of web server VMs, each of which is delivering 2KB web pages. Denali utilizes the full disk swapping bandwidth for random requests. For heavy-tailed request distributions, Denali’s performance surpasses raw disk performance by caching popular virtual machines in memory.

number of virtual machines by swapping inactive VMs to disk. Figure 3 demonstrates Denali’s ability to scale up to 10,000 web server VMs. Requests were distributed across VMs according to two distributions: random and Zipfian (heavy-tailed) with $\alpha = .8$; all requests fetched a 2KB web document. For small numbers of VMs, Denali itself is not a performance bottleneck: the performance of the system is a function of the performance of the applications running on top of it. For example, Denali achieves an aggregate throughput of over 5,000 requests per second for up to 500 web server virtual machines (not shown on Figure 3). For large numbers of VMs, Denali’s performance is disk-bound; our three disk subsystem can swap in 7 virtual machines per second. Denali’s performance for random requests utilizes the full disk bandwidth; Denali’s performance for heavy-tailed requests exceeds the raw disk bandwidth by caching popular virtual machines in memory.

4. Future Directions

Isolation kernels are relevant to a wide variety of application domains. In this section, we present avenues of future research for the Denali project. First, we discuss a set of mechanisms that could be added to the Denali isolation ker-

nel to enhance performance or increase its functionality. We then outline several application domains to which isolation kernels can be applied.

4.1. Mechanisms

Performance isolation: Because an isolation kernel runs directly on the hardware and exposes (virtualized) hardware resources, it can precisely account for physical resources consumed by each VM. This fine-grained resource accounting should make it possible for an isolation kernel to enforce *performance isolation* between mutually distrusting applications, as well as security isolation.

Transparent sharing of resources: Isolation kernels achieve high aggregate system performance for large numbers of VMs by keeping popular services in memory while swapping unpopular services to disk. Leveraging copy-on-write techniques to transparently share code pages between different VMs may reduce the memory footprint of the popular services by eliminating redundant physical code pages, improving overall system performance by making it possible to keep a larger number of VMs in memory at a time. This technique should prove especially effective when many services use the same guest OS.

Checkpointing/cloning/migration: Because an isolation kernel is essentially an interposition layer between a virtual machine and the physical resources it consumes, an isolation kernel can observe and collect the full state of a VM. This state consists of the VM's memory footprint and virtual device state. Additionally, because virtual devices on two different physical machines will have the same interface, even if the underlying physical devices are different, a VM can potentially be migrated across heterogeneous physical machines. By leveraging these properties, an isolation kernel can provide checkpointing, cloning, and migration capabilities.

4.2. New Application Domains

Virtual clusters: Isolation kernels introduce the possibility of subdividing a physical cluster into several multiplexed virtual clusters, and dynamically growing or shrinking the amount of physical resources given to each virtual cluster. In this model, multiple virtual machines execute in a virtual cluster; each virtual cluster is mapped onto some number of nodes in the physical cluster. VMs from many virtual clusters can be multiplexed on a single physical machine to enable high resource utilization in the face of bursty request streams.

Virtual clusters inherit the scalability and availability properties of traditional clusters while acquiring new capabilities. As a virtual cluster's load increases, an isolation kernel can migrate VMs within that virtual cluster to physical machines with idle resources. Additionally, virtual clusters can achieve high availability in the face of physical node failures. Through checkpointing, VMs running on

a failed node can be quickly restarted on a different physical node, restoring the membership of the virtual cluster to its original state even though the physical cluster's membership has changed.

Wide Area Infrastructures: Isolation kernels allow hosts to execute untrusted code, which should permit application authors to "push" new network services into third-party hosting infrastructure. For example, this should allow web service authors to push dynamic content generation code into caching and content delivery networks. A related application domain is wide-area network experimentation infrastructure (such as NIMI [15]); isolation kernels should allow multiple potentially untrustworthy experiments to be deployed and executed simultaneously on shared wide-area infrastructure.

Mobile devices: Mobile and wireless devices, such as Palm Pilots and iPAQs, are playing an increasingly important role in our computational infrastructure. Incorporating isolation kernels into these devices provides a mechanism for the safe download and execution of arbitrary code, permitting these mobile devices to acquire and run context-specific application code as they move between environments.

5 Conclusions

In this paper, we presented the design principles behind an *isolation kernel*, a simple, thin software layer that runs directly on hardware, whose function is to subdivide a physical machine into a set of fully isolated protection domains. We argued that several emerging applications would benefit from isolation kernels, as they either require the isolation of untrusted code, or the ability to precisely control and account for physical resources across competing software services. These emerging applications include pushing new Internet services into third party hosting infrastructure, deploying dynamic content generation code in content delivery networks, or pushing context-specific applications into mobile devices as they migrate across physical environments. We briefly described the design and implementation of the Denali isolation kernel, and presented experimental results that show that we can scale up to a very large number of simultaneous protection domains, as would be necessary for many of our targeted applications. Finally, we outlined several areas of future research.

References

- [1] M. Accetta et al. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, 1986.
- [2] G. Banga, P. Druschel, and J. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating system design and implementation*, Feb. 1999.

- [3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching, and Zipf-like distributions: Evidence, and implications, Mar 1999.
- [4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, 1996.
- [5] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, Oct. 1997.
- [6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP '01)*, Oct. 2001.
- [7] D. Ely, S. Savage, and D. Wetherall. Alpine: A user-level infrastructure for network protocol development. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems (USITS '01)*, March, 2001.
- [8] D. Engler, M. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.
- [9] B. Ford et al. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.
- [10] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the sixth USENIX Security Symposium*, July 1996.
- [11] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1972.
- [12] R. Goldberg. A survey of virtual machine research. *IEEE computer magazine*, 7(6), June 1974.
- [13] D. Gollmann. *Computer Security*. John Wiley and Son, Ltd., 1st edition, Feb. 1999.
- [14] K. Lawton. Running multiple operating systems concurrently on an IA32 PC using virtualization techniques. <http://www.plex86.org/research/paper.txt>.
- [15] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large-scale internet measurement. *IEEE Communications Magazine*, 36(8):48–54, August 1998.
- [16] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1975.
- [17] B. Schneier. *Secrets and Lies: Digital Security in a Networked World*. John Wiley and Sons, Aug. 2000.
- [18] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 Annual Usenix Technical Conference*, Boston, MA, USA, June 2001.

Security Architectures Revisited

Hermann Härtig

Technische Universität Dresden
haertig@os.inf.tu-dresden.de

Abstract

The knowledge in technologies needed to build secure platforms, or Security Architectures, has significantly matured over the recent years. These include small interface technologies, access-control contracts, tunneling, secure booting, effective resource control, and virtual machines. Putting together these ingredients into a small secure platform seems straightforward, yet still remains to be done, and has the potential of making operating systems more dependable.

1 Introduction

In the last decade, several (operating) system projects were well underway to build platforms for applications with very high security requirements. Examples include DSSA (Digital Systems Security Architecture [5]), Trusted Mach [2], and BSA (BirliX Security Architecture [7]). None of them was used in practice, at least not in notably large scale.

One reason may be the (right or wrong) perception that such systems were and — despite the events on September 11th — are not needed. Another reason may be that the technology to build such systems was not mature enough at the time.

This paper claims that today the knowledge to build such a platform is well established in the operating-systems community and that the required technologies have significantly matured over the recent years. However, they still need to be combined into a proper architecture. The remainder of the paper shortly states my view of the requirements for a small secure platform and discusses the needed technologies and their status. It finally puts these ingredients together into a general-purpose small secure platform and explains the use in a dedicated embedded system.

2 Requirements

Devices such as mobile phones or PDAs, but also notebooks and desktops will be and actually are used for valuable or critical applications such as banking transactions, while on the same device all sorts of potentially dangerous rubbish applications are downloaded from the Internet.

Embedded systems are increasingly connected to and controlled via the Internet. In comparison to such systems, servers have advantages since they are (at least should be) in a physically controlled and carefully administrated environment. But also for servers, mobile code — intended as such or unintended — presents a severe threat. The prime requirement for these systems is that the valuable components and applications are reliably protected from the other parts.

To achieve this protection, a careful implementation of a small secure platform (to avoid the infected term of a trusted computing base) is needed that

- provides minimal yet sufficient functionality for applications with high security requirements,
- is flexible enough to be used either as complete general purpose platform, as a scaled down dedicated embedded system, or as a thin server,
- enables in real practice the employment of the *principle of least privilege*, especially but not only for mobile code, and uses it in the architecture,
- supports separation of secure and insecure parts of the system to an extent that even the successful penetration of the core of the insecure part does not endanger the secure side,
- can prove its identity to near or far away communication partners,
- provides compatibility for legacy applications, is an open (in contrast to a closed language) system, and supports reuse of potentially insecure components (e. g., of a network protocol stack), and
- is still small and simple enough for thorough evaluation.

Small and simple enough in more precise terms is in my view: A small group of people, for instance around seven, must be able to completely control the secure platform, i. e., each member of the group should understand all interfaces of the components of the architecture, and each component should be completely understood by at least one member of the group. Systems of the size of the Linux kernel, not to speak about recent versions of other desktop operating systems, will have tough times to achieve this property.

3 Technologies

The technologies needed to build a small secure platform are small-interfaces technologies, tunneling, secure booting, access-control contracts, effective resource control, and virtual machines. We discuss each shortly and point out advances and limitations of their current state of the art.

3.1 Small-interface technologies

For a secure platform to be under complete control of a small group, it must be built as a small set of small components with small interfaces. An example for a large interface is interaction of components by unrestricted usage of pointers in a shared address space, still the most favorite way to build large, monolithic operating-system kernels.

In recent years, the operating-systems community developed two (competing) technologies to build systems with small interfaces. Both are based on small kernels that are supposed to be small enough for thorough evaluation. Then systems are extended either by adding extra functionality on top of these kernels and in their own address spaces or by downloading extra functionality in safe ways into these kernels. The former is usually referred to as microkernel-based approach while the latter is called the extensible-systems approach.

In the **microkernel-based approach**, small interfaces are enforced by separate address spaces, which are effective even in the presence of pointers. Unsafe components can be isolated in their own address spaces. The long held perception of microkernel-based systems to be extremely slow has been proven wrong by work based on the L4 family of kernels [11, 6]. However, the applicability of these results to systems with really small interfaces still needs to be investigated because these current systems use large components (single-server Linux) and make use of shared regions of memory.

A difficult problem of the microkernel approach comes with input-output drivers, if unrestricted use of DMA is allowed. Then, a malicious component, though encapsulated in its own address space, may initiate DMA transfers to any physical address, thus in fact breaking the encapsulation via address spaces. We know of two approaches to tackle that problem:

The first one is to disallow DMA for untrusted components. One technique to do that is to rely on virtualization of hardware which in practice can be done for a limited number of devices only. The other technique is to mediate all DMA accesses through a trusted server. Again, current PC hardware with its not-exactly-systematic DMA interfaces makes it necessary to look at each driver separately. The situation becomes even harder for programmable devices.

The second approach is to restrict DMA access to a partition of physical memory by hardware. This is simple if a bus controller supports this. Then, all DMA, including that

from trusted components, first goes into that partition and then is copied to the trusted component's address spaces. The content has to be protected by other means (see next subsection on tunneling techniques). To solve this problem more thoroughly, additional help may be needed from hardware designs by providing a notion of address spaces on devices or bus controllers (sometimes referred to as DVMA). This may come with as little effort as a riser card for PCI devices [17].

In the **extensible-systems approach**, small interfaces are enforced by restricting the components to be downloaded into the kernel. A common restriction is the use of safe languages [3], which implies inherent difficulties in dealing with input-output drivers. Another approach is to include a transaction-like mechanism into the kernel [15]. However, this seems to cause even harder performance problems than those of the microkernel fraction. A third line to mention is having pieces of code carrying their own proofs which are checked before activation [12].

Both technologies are developed far beyond the status of the ones available to the builders of the previous attempts to build security architectures. As an example, the Mach kernel used as base of the Trusted-Mach architecture, was more than ten times larger (interface, size, cycles, ...) than L4.

In addition to supporting small interfaces, both technologies have the potential to provide legacy operating-system interfaces to support legacy applications, a requirement listed in Section 2. For instance, to move the Linux kernel onto user level as a binary-compatible single-server emulation of the Linux system-call interface required changes respectively additions of around 7000 lines of code. Performance comparisons between a Mach-based and an L4-based single-server implementation of the Linux kernel bring home the point that the technology has matured significantly [6].

3.2 Tunneling

Tunneling is a technique to use software that by itself does not provide a required property and adding this property in an additional layer. Well-known examples are using an insecure protocol for secure communication by encrypting packets before handing them over or tunneling IPv6 packets over IPv4 channels. The term tunneling as used in this paper may be overextending the more common uses of this term.

Potential applications of tunneling for a small secure platform include the file system and input-output drivers:

The **file system** does not need to be part of a small secure platform unless denial of service is a concern. Instead, an untrusted file system can be used to store encrypted information. This implies for a security architecture that an existing file system can be reused, for example the file sys-

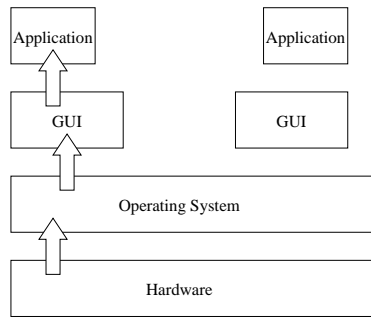


Figure 1. Authentication chain

tem of a complete Linux system in the insecure part of the architecture. Of course, an adversary can destroy data in the Linux file system after a successful intrusion, but can neither obtain access to confidential information nor apply unnoticed changes. This may be completely acceptable if a local file system merely acts as cache for a remote but accessible file server.

Still, the ability to protect some storage such as a minimal flash file system for cryptographic keys needs to be part of the minimal secure platform. If enough such protected storage is available, it may make sense as well to store data that has been modified since last backup, which turns the file system into more than just a cache.

Input-output drivers increasingly become the most complex, ugly, and least-controlled parts of systems. Their partial removal from the trusted part would enormously increase the possibilities for a thorough evaluation of a platform. Though nobody tried this, it seems possible, if and only if DMA management can be put under control. For instance, a trusted file system does not need a trusted disk driver.

The inherent limitation of tunneling however is in denial of service attacks. If a driver is needed for a requested functionality, it must be part of the secure platform. But in embedded appliances, where Internet access is often needed for reporting purposes only, protocols can be left out of the secure platform and reused using tunneling techniques. Hence, an aggressive combination of small interface technologies and tunneling indeed promises to have the potential of keeping the small platform small enough to stay under complete control of a small group of developers.

3.3 Secure booting

Neither microkernels nor tunneling solve the problem of What You See Is What You Get. It is easy to emulate a user interface or a complete device that pretends to be something which it is not, hence prompting the user to unveil secrets involuntarily. A recent successful attack on PGP made use of exactly this phenomenon.

The technique to solve this type of problem has been known for over 10 years [5, 7] as *secure booting*. Secure

booting ensures that a specific hardware with a specific OS with a specific GUI and a specific application is indeed running in the identified device. Secure booting relies on hardware to establish the identity of a boot loader, on the boot loader of the operating system, on the OS of the user interface, and so on, thus forming a bottom-up chain of authentication rooted in hardware (Figure 1). An authentication protocol can then be used to verify this chain from a remote computer. A remote computer may be a server several thousands of miles away or a smart card used to locally identify a device.

Recent attacks on cell phones can be attributed to this class of problems. There, cheap cell phones were sold under the precondition that for a certain limit of time only a certain provider can be used. It was part of the operating systems' functionality to enforce this deal. The attackers stole the cell phones and replaced the operating systems. Their communication partners were not able to find out whether or not the cell phone was using its original operating system.

There are two limitations that need to be mentioned. One is the need for physical protection of the devices, notably their tamper resistance and freeness of side channels. Side channels are means to extract confidential data such as cryptographic keys using *unintended interfaces*. An example for an unintended interface is power consumption that changes depending on whether a 1 or a 0 is currently processed in a cryptographic key. An obvious unintended interface is direct access to the memory bus of a device, for instance with in-circuit emulators. Tamper resistance requires — in contrast to tamper proofness — that unnoticed modifications are impossible (or very unlikely). Both avoidance of unintended interfaces and tamper resistance are hard to achieve, depending on the efforts of the attacker. A constant flow of news from the University of Cambridge impressively brings home that point.

The other limitation against which secure booting does not defend is an attack that is sometimes referred to as the *Mafia Fraud*. Here, an adversary replaces a device with a faked one that forwards all communication to the original one and thus successfully performs all required protocol steps of the secure booting protocol. It then can show an arbitrary user interface that prompts the user to reveal secrets involuntarily. Several techniques to solve this problem have been proposed. The most promising in my opinion is based on frequent, key-controlled hopping between large numbers of channels. This solution will be presented at the upcoming military communication conference [1].

The apparent change within the past years is that manufacturers started building devices based on such ideas.

3.4 Access control based on contracts

In practice, access rights in end-user systems are granted much too lavishly. The main reason for that — besides neg-

ligence — is the complexity of mapping an intended security policy to discretionary access-right mechanisms, at least as provided by today's platforms. Rule-based schemes did not help in practice either, since the known sets of rules turned out to be of not much use outside of the domain of military document systems. In other words, an intuitive and practical way of correctly employing the principle of least privilege is missing. In the increasing presence of mobile code — some prophets are even foreseeing mobile code as the governing principle for software usage — this situation is dangerous.

Help may come from lifting the process of granting access rights up one level of abstraction. Access rights then are based on contracts of the following kind: *If I get access right to these specific resources, then I will perform that specific function for you.* The requested access rights have to be shown up front. For example, before a new program is installed, it needs to show the contract, i. e., which resources it needs to perform which function. Then, before an installed program is executed, it explicitly presents the access rights to the objects it needs beyond the access rights already granted to the installed program, if any. As another example, before a new piece of mobile code is downloaded and started, it needs to state its needs and promised function.

Several implementations of this scheme have been published. Even UNIX can be seen as having an early version of this: Manual pages used to contain a section “used files” stating the resources needed, yet were used for documentation only.¹ All of these implementations share most of the techniques and properties, but also at least one major hard problem that needs to be solved:

Programs in general and mobile code in particular become offers for contracts, i. e., together with actual code and identification of vendor, each program contains a (symbolic) list of resources needed. To ensure this mapping of code to vendor and requested resources, the integrity of contracts must be enforced by cryptographic signatures. Whenever a program is installed, the list of requested resources is shown up front and offered to the user for approval.

Constantly being requested to approve contracts will certainly lead to the *too-many-alarms effect* and in consequence to unintended approvals of contracts. Hence, shortcuts are needed. One way to provide shortcuts is the use of (symbolic) access-control lists (ACLs) expressing trust with regard to sources of code. For example, all programs of a certain distribution or package are trusted for a specified set of objects. Whenever a new process is started, the requested rights are first checked against these ACLs before confronting the user.

The result of an agreed-upon contract is a (symbolic) list

¹This analogy was pointed out by Sape Mullendar when explaining how the Amoeba architects intended to generate capabilities from machine readable documentation.

of resources the process (the program in execution) may use, in principle similar to a (classical and old-fashioned) capability list. Accessing an object, for instance opening a file, without possessing the proper capability is a violation of the contract. In contrast to classical capability and ACL schemes, capabilities and ACLs are in symbolic form, e. g., in some platform-independent form that can be mapped to system-dependent names, for instance in a Unix-like system to file or DNS names or to symbolic representations of parameters. These capabilities then can and must be inspected at all invocations that carry symbolic names, for instance in a Unix-like system at open, connect and bind, and various exec system calls. Classical capabilities containing non-symbolic identifiers of object and access rights to them, for instance Unix's file descriptors, are created by some of these invocations and are used and checked thereafter. Adding enforcement of symbolic capabilities to a system like Linux requires about 350 lines of changes to the kernel [10].

Capabilities, at least at the symbolic level, may have to include parameters. For instance, to control access to a modem, it makes sense to include the requested telephone numbers. Again, this is simple if a symbolic command-line interface such as often used to invoke Unix programs is enforced for accesses to resources in question.

An example for a symbolic requested resource is “*dial 1 800 123 4567891*” which is accepted if an ACL contains “*dial 1 800 **”. Sometimes, mappings are expressed explicitly, for instance a program may request access to “*mailer-spool-dir*” and the corresponding ACL may contain “*mailer-spool-dir is /var/spool/mail*”.

From our experience, the **hard problem** in practice is that all interpreters must be made to obey access-control contracts. For example, a JVM running in a network browser may either implement it by itself² or the underlying operating system may force it to do so. The former requires trust in all interpreters, and there are (too) many, as the spread of macro viruses via document editors shows. The latter requires the interpreters to be modified such that for each new contract (e. g., a downloaded applet) a new encapsulated entity of the underlying system (e. g., a new process running the browser in Unix-like systems) needs to be created. The contract, the resources needed by the new browser process on behalf of the newly downloaded applet, needs to be forwarded to the underlying operating system and attached to the newly created encapsulated entity (Unix process). This problem becomes more complicated for more interesting forms of interpreter nesting. The attempts of several of my bravest students to do so with the Netscape browser failed miserably.

²IBM's Flexxguard System implements a similar scheme.

3.5 Effective resource control

Effective defense against denial-of-service attacks through blocking of resources needs effective resource control. Work in the real-time systems community has brought significant progress in two areas.

One of them allows better control over the allocation of resources to parts of a system. Systems such as Resource Kernels [14] and DROPS [8] in principle allow reserving resources independent of their type, i. e., CPU as well as disk bandwidth, for example. While, to my knowledge, such systems are still very much in prototypical status, their potential is promising.

The other area is a technique called early demultiplexing. While in classical implementations of protocols, the decision on whether or not to throw away a packet came rather late and thus wasted resources just for policing, newer implementations even making use of small dedicated CPUs in off-the-shelf network-interface cards have proven the progress. Examples with numbers are given in [4].

Both techniques seem helpful in preventing localized denial of service attacks based on intentional resource exhaustion. Dealing with distributed attacks will require support from routers or lawyers.

3.6 Virtual machines

Virtual machines support legacy at an even lower level in comparison to the emulation of operating system interfaces. Legacy operating systems run (nearly) unchanged. This replaces the n -fold effort to emulate n legacy operating-system interfaces by the emulation of just one hardware architecture. This technique, pioneered by IBM's mainframes, became available for PC architectures in recent years.

Isolation or separation is supported by providing different machines for different (classes of) applications. Especially, it solves the DMA problem mentioned in Section 3.1 by emulating input-output devices such that even malicious drivers cannot break the separation (as is possible in current small-interface technologies). However, this comes at the cost of emulating the devices, which is higher than the cost involved with identifying and controlling their DMA accesses.

3.7 More on technologies

This paper concentrates on technologies in the operating-systems domain. However, it must be stated that software-engineering technologies to systematically build more reliable software in general have matured as well since the last significant efforts to build security architectures.

Methods for static analysis to discover certain types of faults have advanced. C (and C++), the most favored language(s) for operating-systems builders, have never earned

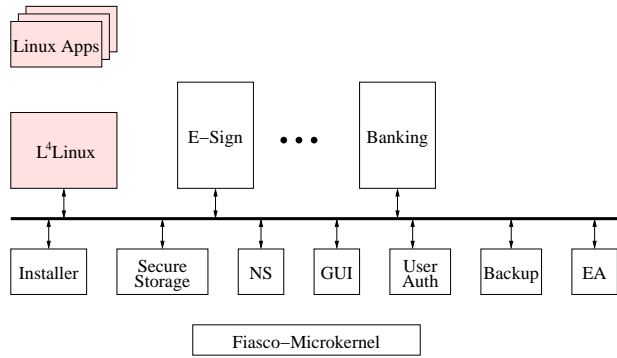


Figure 2. The Nizza architecture

the reputation of being the most advanced language with respect to safe programming. Now, some restricted versions of C are proposed that maintain its versatility, but allow stricter analysis.

These and related technologies of good software engineering and programming practice are considered as orthogonal to the architectural technologies discussed in this paper. I ignore them henceforth.

4 Architectures

Putting together these technologies into a security architecture based on a small secure platform seems rather straightforward in principle. Still, this section should be taken with a ton of salt, since so far it is educated speculation to a very large extent. We sketch a rather general-purpose platform (Nizza) and an application of the technologies to a dedicated system (Micro-Sina).

4.1 Nizza

Nizza³ is a back-of-the-envelope (literally speaking) design for a small secure and general-purpose platform supporting applications with high security requirements such as digital signatures and banking protocols while still having the option of running legacy code. It is sketched in Figure 2. Discussing Nizza serves as an opportunity to identify the critical components and the role of technologies discussed before.

The architecture separates the system into two parts, the legacy and potentially rubbish part (left) and the secure part. Both, legacy and secure sides, run on the small secure platform whose principle tasks are to provide minimal sufficient functionality for applications with high security requirements and to ensure separation. Applications on the secure side reside in their own address spaces. They contain all functionality which is needed but not provided by the small secure platform, e. g., as libraries. New applications on the secure side can be installed from outside by

³Nizza is the German name of Nice, France

contacting the legacy side which forwards such requests to the installer. This also is the first usage of tunneling: Rather than providing transport and higher-level protocols in the secure side, the insecure side's functionalities are used.

The unsafe side, L⁴Linux and its applications, is based on a user level implementation of the Linux kernel [6]. It uses transparent libraries as invented by Mach for binary compatibility for most applications. L⁴Linux is running in its own set of address spaces.

The prime requirement for the insecure side is that it cannot harm the secure side, even if its core, the former *Linux Kernel*, is penetrated successfully. L⁴Linux — on an as-it-is basis — needs to be *tamed* in several aspects:

- L⁴Linux needs to be modified to have the X-Windows server run on a frame-buffer implementation that is provided by a secure GUI (see below).
- Input-output drivers that need to be part of the small secure platform need to be taken out from L⁴Linux, however can still be used via stubs from L⁴Linux. This technique is reasonably well understood due to the reuse experience of drivers in the DROPS real-time system.
- DMA needs to be put under control to enforce address-space separation. If hardware support is available to restrict DMA to a partition of physical memory, all DMA — including the one initiated on the secure side — is directed to that partition and then copied to secure memory. Hence the contents must be protected by other means, e. g., via tunneling techniques. The effort needed for that and the resulting performance remain still unclear.

Having a virtual machine instead of L⁴Linux would be desirable since it is more general with regard to legacy software. Then however, protocols to cross machines boundaries (e. g., TCP/IP) belong to the secure part.

The platform is based on Fiasco, a careful reimplementation of the L4 interface. It provides address-space separation as a basic means to keep interfaces small. The L4 microkernel interface has about a dozen system calls, hence the interface can be considered small. It is not only under complete control of one person, but is small enough to have triggered Dresden's theory group to leave alone their stacks and lists and try a source-code-based formal verification of a claim that is fundamental for address space separation: "Only kernel-code runs in kernel mode" [9]. The major drawback of the current situation is L4's elegant but notoriously inflexible chiefs&clans mechanism [11]. Considerations are well under way to replace it by somewhat else in one of the future versions of the interface.

An important component of the small secure platform is the name server. It provides a symbolic name interface to all resources of the small secure platform including a communication interface to the insecure side. It maintains the

certificates needed to check the validity of requests from outside. An important role of the name service is to provide symbolic names that are used in access-control contracts, capabilities, and ACLs.

The "installer" is the component responsible for loading and installing other components of and new applications on the secure side of the system. The installer's responsibility includes the decisions on newly-proposed access-control contracts and the derivation of symbolic access-control and capability lists. The installer however needs to take care only of the small secure platform as an *interpreter*, not of nested interpreters. An installer is a complex component, alone due to the machinery needed to load and establish applications besides and independent of L⁴Linux. However, it can rely on L⁴Linux to load programs from servers. The installer needs to be aware of secure booting. It establishes the authentication chain and makes it available to the trusted GUI. We will subsume the boot loader and related components to be part of the installer and will not discuss that part of secure booting henceforth.

The secure storage component, if it is designed for confidentiality and integrity, again can rely on tunneling. The actual storage of large data can be left to the L⁴Linux file system. Encryption technology can be used to protect data against information dissemination and unnoticed modifications. It cannot protect against destruction of data, i. e., against a denial of service class attack. However, this may be tolerable if almost all data on a PDA or cell phone is stored on a server anyway reducing the secure storage component to a mere cache for the largest part of data. However, direct, i. e., untunneled secure storage is needed for the keys and for data added since the last backup. It seems, that soon modern chips can have enough memory on chip to avoid having to add an extra external module to the tamper-resistant device.

A trusted GUI component must reliably show the authentication chain of the application that is currently controlling the screen. To this end, it needs to provide an interface to the X window system that disallows direct access to the video memory. The component to authenticate human users need to be part of the small secure platform as well. Keeping input-output drivers in the secure platform small, for instance by reusing existing drivers with tunneling techniques, will be a major challenge.

We assume that everything else can be done at the application level. This includes cryptographic infrastructures as needed by specific applications.

4.2 Micro-Sina

Micro-Sina⁴ is an effort to replace a Linux-based implementation of a VPN box by one based on the Fiasco

⁴Micro-Sina is sponsored by the BMWi and done in cooperation with Secunet AG.

microkernel. The objective is to identify and implement the minimal functionality that is needed for that purpose. Micro-Sina will (probably) not include secure booting since physical protection is provided for these boxes. It will contain a secure storage component, a name server, and some input-output drivers. In notable contrast to Nizza, L⁴Linux can (probably) not be used for TCP/IP- or IPsec-tunneling. Reusing L⁴Linux's IPsec implementation would require trust in L⁴Linux, and reusing its TCP/IP implementation after having done the encryptions would violate IPsec's data-format obligations. Hence, according to our current understanding, we need to extract Linux's (or another system's) IPsec implementation and carefully port or completely rebuilt it from scratch on top Fiasco as part of the small secure platform. This is an example for a major limitation of the applicability of tunneling.

5 Related work

For related work, we will concentrate on security architectures, i. e., on the integrative usage of the key technologies rather than adding more references to the technologies per se. We will skip closed-language-based systems and real-time systems that employ separation techniques in similar ways (such as Oncore System's and DROPS' real-time variants of Linux). In my view, IBM's and VMWare's implementations of virtual machines and completely new implementations such as the EROS operating system come closest to what I claimed to be desirable. However none of these comes anywhere near to some derivatives of MULTICS and to the very well-written descriptions of the DSSA. The most complete integrative implementation of a Nizza-like architecture is probably Christian Stübke's Perseus [13].

5.1 Classical security architectures

DSSA [5] is centered around an elaborate authentication scheme which is rooted in hardware and extends up to the authentication of application processes. It is used in combination with very flexible ACLs that allow fine-grained authorization. Although DSSA has never been implemented completely, the authors claim credibly that they did not encounter difficulties that they suspected to be unsurmountable. DSSA was accompanied by a technique that allowed formal reasoning about the validity of claims of their protocols. Though these formalisms met scepticism in the cryptography community, it at least turned out a powerful tool in convincing peers about certain claims (as I had experienced once when being the object to a such as exercise). DSSA did not look at other techniques discussed in this paper. It did not care about the size of implementations of secure platform. Although the ACL scheme was fine-grained, there were no considerations about how to use them to express users' needs.

Trusted Mach [2] (T-Mach) is a careful implementation

of multi-level security. It is based on a kernel derived from the Mach microkernel that implements a reference monitor for a multi-level security policy. T-Mach certainly suffered from the then state of the art in building small kernels. A fair appreciation of T-Mach based on the available information and the space available for this paper is not possible.

BirliX was a fairly complete, object-based reimplement of the Unix kernel interface in a language safer than C. The effort needed to achieve true binary compatibility turned out enormous which leads to my perception that reuse of off-the-shelf operating systems for the insecure side is the method of choice. BirliX had reinvented secure booting but did not implement it either. It had a notion of combining capabilities (*subject restrictions* in BirliX speak) with ACLs of fine granularity. Subject restrictions were carried around by programs, coming close to but not quite arriving at access-control contracts.

5.2 Virtual-machine implementations

In discussions with strict believers in virtual-machine technology I observed the somewhat naive perception that virtual machines solve all problems radically per se.

They do not. First, separation of machines does not necessarily separate applications. It does not make a big difference whether network or local IPC is used to allow applications to communicate with the outside world. Some way of controlled interaction and installation of applications must be provided, e. g., access-control contracts. Second, the malicious-driver problem is solved only for drivers for emulated hardware devices, not for the drivers needed to implement the virtual machine. Hence, the small-platform problem that becomes hard when input-output devices are part of the platform still needs to be solved, best using some small-interface technology. Once a device is so well under control that it can be emulated, it is fairly easy to make sure the driver does not use DMA to corrupt other processes address spaces. Hence, the problem of providing encapsulated drivers remains hard. Third, virtual machines do not solve the *small* platform problem per se although building a small virtual-machine implementation seems possible. Fourth, the secure-booting problem is orthogonal to using virtual machines. Fifth, it does not help to provide virtual machines if the operating systems running on them are not secure.

However, some of the problems described in this paper are well addressed by some of today's virtual-machine systems. LPAR, IBM's implementation of virtual machines in their z-Series line of machines, is small enough to have earned an EAL-5 evaluation. Also, VMWare's new server-oriented implementations are not based on Linux or Windows 2000 anymore, but on a smaller kernel. Providing a virtual machine is certainly the technology of choice to support legacy.

Remains to say: *If only all CPUs would clearly separate user/kernel from metal/virtual issues to support efficient implementations of virtual machines (unlike the x86 architecture).*

5.3 EROS

EROS [16] — an operating system that has been developed from scratch — is based on a carefully designed capability system and persistent single-level storage. EROS addresses several problems mentioned in this paper.

EROS certainly possesses the basic security mechanisms to provide access-control contracts. It does not (yet) address the problem of cooperating interpreters.

The authors claim that a Linux-compatible environment is in progress, but from our experience with BiriX we assume that this effort is in danger of becoming a never-ending hunt for a moving target. For legacy applications, we see no other practical chance than either virtual machines or adaptation and encapsulation of original operating-system implementations to a new underlying small secure platform.

For EROS, critical applications need to be refactored from existing ones into components to take advantage of the underlying kernel's security properties, which induces a significant development cost. EROS' authors do not seem to look in systematic application of tunneling techniques to save effort for the implementation of the secure partitions of the platform.

EROS postulates the implementation of distinguishable trusted and untrusted user interfaces but does not take into account adversaries that can replace EROS by another operating system. It does not address secure booting techniques to prevent that kind of attacks.

Overall, EROS, to our knowledge, is a well-designed capability system that addresses some but not all problems of this paper and uses some but by far not all technologies that are at hand for the design of secure architectures. As a side remark, we expect the anticipated effort to gain EAL 7 certification will be hard, since EROS is not to be based on a small-interface technology.

6 Acknowledgements

The envelope used for designing Nizza laid on a table on Nice's beach surrounded by Birgit Pfitzmann, James Riordan, Michael Waidner, Arnd Müller, and myself. Christian Stübke, PhD student at Saarbrücken, then started to undertake the brave attempt to bring the envelope to paper and to start implementing some of these ideas.

Many discussions with students and other members of the operating-systems and real-time group of Technische Universität Dresden helped a lot when writing this down. Michael Hohmuth and Christian Stübke have helped in finishing this paper.

Dresden's operating-systems and real-time group gratefully acknowledges generous grants from DFG, BMWi, IBM, Intel, and others that enabled this work.

7 Conclusion

The topic of this workshop is *Can we depend on OSEs?*

The answer is: *We could much more so, if only the technologies that have significantly matured over the recent years would be put to proper use.*

References

- [1] A. Alkassar and C. Stübke. Towards secure IFF - preventing mafia fraud attacks. Accepted for IEEE Military Communications Conference 2002 (MILCOM), Anaheim, California, Oct. 7-10, 2002.
- [2] N. Associates. Trusted Mach — specifications. URL: <http://www.nai.com/research/nailabs/finished-projects/trusted-mach.asp>
- [3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the Spin operating system. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 267–284, Dec. 1995.
- [4] U. Dannowski and H. Härtig. Policing offloaded. In *Proceedings of the Sixth IEEE Real-Time Technology and Application Symposium*, Washington D.C., May 2000.
- [5] M. Gasser, A. Goldstein, C. Kaufmann, and B. Lampson. The Digital distributed system security architecture. In *12th National Computer Security Conference (NIST/NCSC)*, pages 305–319, Baltimore, 1989.
- [6] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ -kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Oct. 1997.
- [7] H. Härtig, O. Kowalski, and W. Kühnhauser. The BiriX security architecture. *Journal of Computer Security*, 2(1):5–21, 1993.
- [8] H. Härtig, L. Reuther, J. Wolter, M. Borriß, and T. Paul. Cooperating resource managers. In *Fifth IEEE Real-Time Technology and Applications Symposium (RTAS)*, Vancouver, Canada, June 1999.
- [9] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März 2002, Dresden University of Technology, 2002. URL: <http://os.inf.tu-dresden.de/vfiasco/>
- [10] S. Lehmann and A. Westfeld. Kapselung ausführbarer Binärdateien. slcaps: Implementierung von Capabilities für Linux. In D. Fox, M. Köhntopp, and A. Pfitzmann, editors, *Verlässliche IT-Systeme (VIS)*, pages 21–35. GI, Vieweg, Sep. 2001.
- [11] J. Liedtke. Toward real μ -kernels. *Commun. ACM*, 39(9):70–77, Sept. 1996.
- [12] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, 1996.
- [13] B. Pfitzmann, J. Riordan, C. Stübke, M. Waidner, and A. Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research Division, Zurich Laboratory, Apr. 2001.
- [14] R. Rajkumar, C. Lee, J. Lehoczyk, and D. Siewiorek. A resource allocation model for qos management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997.
- [15] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 213–227, Seattle, WA, Oct. 1996.
- [16] J. Shapiro and N. Hardy. EROS: A principle driven operating system from the ground up. *IEEE Software*, pages 26–33, Jan. 2002.
- [17] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A pre-silicon software development environment for the IA-64 architecture. *Intel Technology Journal*, (4), 1999.

The Case for Transient Authentication

Brian D. Noble and Mark D. Corner

Department of Electrical Engineering and Computer Science

University of Michigan, Ann Arbor, MI

{bnoble,mcorner}@umich.edu

Abstract

How does a machine know who is using it? Currently, systems assume that the user typing now is the same person who supplied a password days ago. Such persistent authentication is inappropriate for mobile and ubiquitous systems, because associations between people and devices are fleeting. To address this, we propose transient authentication. In this model, a user wears a small hardware token that authenticates the user to other devices over a short-range, wireless link. This paper presents the four principles of transient authentication, our experience applying the model to a cryptographic file system, and our plans for extending the model to other services and applications.

1 Introduction

How does a device know that the person using it is the right person? Unfortunately, authentication between people and their devices is both *infrequent* and *persistent*. Should a device fall into the wrong hands, the imposter has the full rights of the legitimate user while authentication holds.

To see why, first consider how two computational principals authenticate one another's messages. Each principal knows some shared secret, a *session key*. The sender uses this key to compute a *message authentication code*, or MAC, which is embedded in each sent message [11]. The receiver recomputes the MAC to verify that the presumed sender did in fact send that particular message. Each message is inseparably bound with the proof of its authenticity; authentication is *atomic*.

Unfortunately, it is infeasible to ask users to provide authentication for each request made of a device. Imagine a system that required the user to manually compute a MAC for each command. Instead, users authenticate *infrequently* to devices. User authentication holds until it is explicitly revoked, though some systems further limit its duration to hours or days—it is *persistent*.

Persistent authentication has been acceptable for personal computing because PCs have relatively strong physical security. For example, it is likely that the person typing at the keyboard of my office workstation is someone that I trust. However, mobile devices are easily carried, and therefore easily lost or stolen; if someone steals your laptop while you are logged in, they have full access to your data. Likewise, ubiquitous computing elements are often public, accessible to trusted and untrusted users alike.

One way to limit the vulnerabilities of persistent authentication is to limit its duration. This increases the user's burden, encouraging him to disable security entirely. By default, Windows 2000 asks users to reauthenticate whenever a laptop awakens from suspension. Anecdotally, we find that many people disable this feature, forfeiting its protection for ease of use.

Persistent authentication creates tension between protection and usability. To maximize protection, a device must constantly reauthenticate its user. To be usable, authentication must be long-lived. We resolve this tension with a new model, called *transient authentication*.

In this model of authentication, a user wears a small token, such as the IBM Linux watch [9], equipped with a short-range wireless link and modest computational resources. This token is able to authenticate constantly on the user's behalf. It also acts as a proximity cue to applications and services; if the token does not respond to an authentication request, the device can take steps to secure itself.

At first glance, transient authentication merely seems to shift the problem of authentication to the token. However, mobile and ubiquitous devices are not physically bound to any particular user; either they are carried or they are part of the surrounding infrastructure. As long as the token can be unobtrusively worn, it affords a greater degree of physical security.

We first enumerate the principles underlying transient authentication. We then briefly describe a proof-of-concept cryptographic file system that uses this authentication model, and our plans for developing an API to support a broad range of services and applications.

2 Transient Authentication Principles

Transient authentication consists of four properties. First, users must hold the sole means to access resources on the device. Second, the system must impose no additional usability burdens. Third, the mechanisms to secure and restore sensitive data on a mobile device need be no faster than the people using it. Fourth, users must give explicit consent to later actions performed on their behalf.

2.1 Tie Capabilities to Users

The ability to perform sensitive operations must reside with the user rather than his devices. For example, the keys to decrypt private data must reside on the user's token, and not on some other device. Each protected application and service must be structured in such a way as to depend on capabilities that reside on the token.

At the same time, it is unlikely that the token—a small, embedded device—can perform large computations such as bulk decryption. Furthermore, requiring the token to perform cryptographic operations in the critical path of common actions will lead to unacceptable latency. In such cases, it may be necessary to cache capabilities on a device for performance. Most often, cached capabilities are obtained through a cryptographic operation using keys only on the token. The decrypted capabilities must be destroyed when the user (and his token) leave, and the master capability should not be exposed beyond the token.

The token and device exchange capabilities using a wireless link. The system must provide confidentiality and integrity for these messages. This is ensured by a session key shared by the token and device, used to encrypt each packet, and to generate a MAC.

One could instead imagine a simple token that responded to authentication challenges. This gives evidence of the user's presence, but does not supply a cryptographic capability. An operating system could use this evidence to govern access to resources, data, and services. Unfortunately, this model is insufficient. If the device is *capable* of acting without the token, then an attacker with physical possession can potentially force it to do so. As a simple example, consider file system access control. An unencrypted disk can be removed and inspected on a machine that does not check for the token's presence. An encrypted disk, with the keys stored only on the token, is not subject to the same attack.

2.2 Do No Harm

Investing capabilities with users increases the security of the system. However, increases in security cannot impose increased user burdens. When faced with inconvenience, however small, users are quick to disable or work around

security mechanisms. Not only must the performance of the machine remain unaffected, but additional usability burdens are unacceptable.

Users already accept infrequent tasks required for security. For instance, passwords are used occasionally, usually on the order of once a day. More frequent requests for passwords are perceived as burdensome; a transparent authentication system should impose no more usability constraints than current systems.

Transient authentication must also preserve performance, despite the additional computation increased security requires. As long as this computation is imperceptible to the user, it is an acceptable burden. For example, the Secure Socket Layer (SSL) [5] protocol requires processing time for encryption and authentication. This cost is easily masked by the latency of loading web pages.

2.3 Secure and Restore on People Time

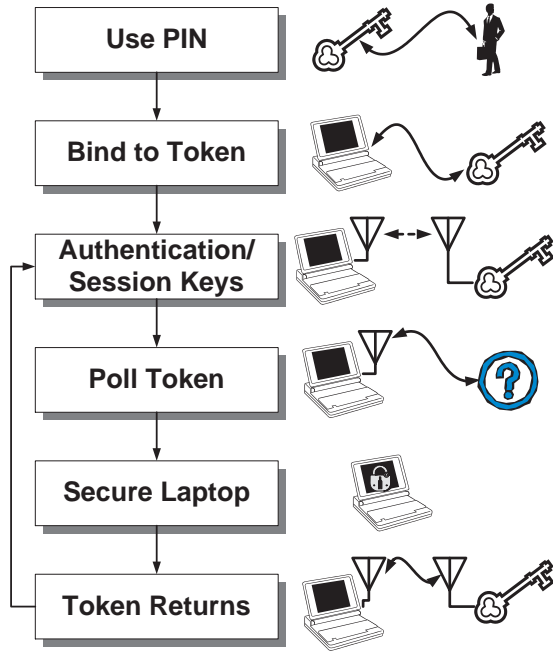
Cached capabilities—and the data they protect—can only remain while the token is present; when the token is out of range, sensitive items must be protected. This process must happen before an attacker gains access to the machine. One might think that this must happen as quickly as possible. However, since people are slow, the limit is on the order of seconds, not milliseconds.

Rather than simply erasing sensitive information, one might prefer to encrypt and retain it. This additional work can save time on restoration: when the user returns, one can obtain the proper key from the token and decrypt the data in place, restoring the machine to pre-departure state. Since the restoration process begins when the user re-enters radio range, it can complete before the user resumes work.

2.4 Ensure Explicit Consent

Tokens and devices must interact securely, and with the user's knowledge. In a wireless environment, it is particularly dangerous to carry a token that could provide capabilities to unknown devices autonomously. A "tailgating" attacker could force a user's token to provide secret keys, nullifying the security of the system. Instead, the user must authorize individual requests from devices or create trust agreements between individual devices and the token.

On one hand, users could confirm every capability requested by the device. However, usability is paramount, thus the granularity of authorization must be much larger. Instead of an action by action basis, user consent can be given on a device by device basis. If this granularity is made smaller, more usability demands would be placed on the user with no corresponding gain in security. For instance, once a sensitive key has been given to a laptop, other programs on the machine can access that key by corrupting the operating system.

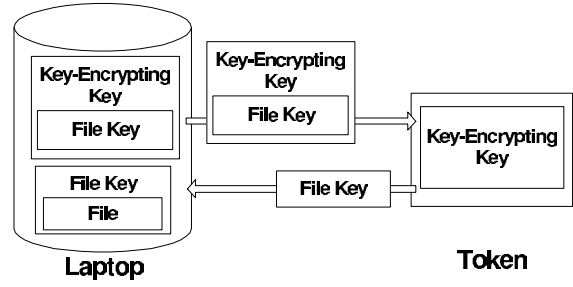


This figure shows the process for authenticating and interacting with the token. Once an unlocked token is bound to a device, it negotiates session keys and can detect the departure of the token.

Figure 1. Token Authentication System

To ensure explicit consent, our model provides for the *binding* of tokens to devices. Binding is a many-to-many relationship; I might interact with any number of devices, and any number of users might share a device. Binding requires the user's assent but can be long-lived. This limits the usability burden. The binding process requires mutual authentication between device and token; this introduces the question of device and token naming, which we have not yet addressed.

Unfortunately it is possible for a user to lose a token. Token loss is a serious threat, as tokens hold authenticating material; anyone holding a token can act as that user. To guard against this, users must periodically authenticate to the token. This authentication can be persistent, on the order of days. Nominally, any authenticating material in the token is encrypted by a user-supplied password; when the authentication period expires, the token flushes any decrypted material, and will no longer be able to authenticate on the user's behalf. Placing authentication material in PIN-protected, tamper-resistant hardware [15] further strengthens the token in the event of loss or theft. The transient authentication process, illustrating all of these principles, is shown in Figure 1.



This figure illustrates the process of file key acquisition. Encrypted file keys are read from disk and shipped to the token. The token decrypts it and returns the file key. All laptop/token communication is protected by a session key, negotiated at bind time.

Figure 2. Decrypting File Encrypting Keys

3 Example: Protecting File Systems

To validate this approach, we have built ZIA, a cryptographic file system employing transient authentication. This prototype demonstrates that transient authentication can protect resources against theft or loss without compromising performance or hampering usability. It is explained in detail elsewhere [3]; we summarize it here. ZIA is implemented as a stackable file system layer [6], utilizing the FiST framework [17].

Tie Capabilities to Users Each on-disk object in ZIA is encrypted with some particular file key, K_f . File keys are assigned per-directory. Since the token cannot provide bulk decryption services, the file key is also stored on disk encrypted by some key-encrypting key, K_k . A file that is shared has its corresponding K_f encrypted by more than one K_k . Tokens hold each applicable K_k ; they are never revealed. Key exchanges are illustrated in Figure 2.

Do No Harm When ZIA receives a read request for a file block, it must decrypt it with the corresponding K_f . If it is not already cached, the encrypted version, $K_k(K_f)$, is sent to the token, which decrypts and returns it. Key acquisition is overlapped with reads, and decrypted keys are cached for later reuse. With these optimizations, ZIA adds an overhead of under 10% for a modified Andrew Benchmark, and just over a factor of two for bulk data transfer. In both cases, the overheads imposed by ZIA are indistinguishable from those of Cryptfs [16], a cryptographic file system with a single key in effect for all files. In other words, the overheads are limited by cryptography, not key acquisition.

Secure and Recover on People Time On token departure, ZIA encrypts each cached file block, and flushes each cached K_f . ZIA does not evict encrypted, cached file blocks, but the OS may later evict them of its own accord. On the largest buffer cache we can observe on our

```

typedef int32_t ta_appid_t;

enum lifetime {TA_LIFE_SESSION, TA_LIFE_PERM};

typedef struct ta_keyname_t {
    char* username;
    char* appname;
    char* keyname;
    enum lifetime life;
} ta_keyname_t;

int ta_initialize();

void ta_shutdown ();

/* Registers an application with the library */
ta_appid_t ta_application_reg(IN char* appname,
                             IN char* username);

```

Figure 3. Library Management Functions

hardware—an IBM ThinkPad 570 with 128 MB of memory, running Linux 2.4.10—this process takes less than five seconds. The window of vulnerability is short enough to foil physical possession attacks. As soon as the token enters radio range, ZIA re-fetches each K_f and decrypts all cached, encrypted file blocks. On our hardware, this process takes less than six seconds. ZIA imposes minimal overhead and preserves usability, giving the user no reason to disable it.

Ensure Explicit Consent Tokens only decrypt file keys for laptops to which they have been bound. When a newly-encountered laptop first asks the token for a key, the token passes this request up to its wearer. The user then decides whether this is a request from a legitimate device. If it is not, the laptop will be ignored thereafter. If it is legitimate, the token and laptop use the Station-to-Station protocol [4] to provide both mutual authentication and session key exchange. This session key is used to protect file key traffic between the two devices until the user departs. When the user departs, the session key is dropped. On return, the prior binding remains in force, so Station-to-Station can negotiate a new key without user involvement. Bindings have a long but finite lifetime, on the order of days.

4 Protecting Services and Applications

It is straightforward to provide transient authentication services to a file system; the semantics of identity and privilege are well-defined, and the implementation is controlled by the operating system. Most applications inherit these same notions of user identity from the operating system. Transient authentication can be extended to applications by protecting the virtual memory space of each process.

By default, the system protects every process on the machine, except essential kernel threads and processes related to token communications. During the boot process, the token and machine agree upon an encryption key. Upon user departure, the operating system suspends each processes and masks arriving signals. The OS then encrypts each in-memory page belonging to the process and erases the key.

When the user returns, the system fetches the decryption key from the token and restores each page. The processes are restarted and the system continues without loss in performance. Combined with swap space protection [13], this mechanism fully protects virtual memory. We are currently implementing address-space protection.

There are two reasons why this brute-force approach may not be desirable. First, completely suspending all applications—even those without sensitive data—is effective but indiscriminate. Second, not all applications inherit their notions of identity and authentication from the operating system. For example, a user browsing the web may interact with dozens of different services, each with its own user name.

We are developing an API for applications to make use of transient authentication services directly. This API includes management functions, shown in Figure 3 and token interaction functions, shown in Figure 4. The initialization and shutdown functions are used to initiate connections to a per-machine service. The registration functions inform this service of the application’s use of the token. This service can then control application requests, including logging and caching of results.

Applications that cache keys—or the decrypted information that they protect—must be informed whenever the user

```

/* Registers/Unregisters callback functions */

int ta_auth_loss_reg(IN ta_auth_hdlr_t hd,
                    IN void *data);

int ta_auth_loss_unreg(IN ta_auth_hdlr_t hd,
                      IN void *data);

int ta_auth_gain_reg(IN ta_auth_hdlr_t hd,
                    IN void *data);

int ta_auth_loss_unreg(IN ta_auth_hdlr_t hd,
                      IN void *data);

/* Decrypt a key with the token */
int ta_decr_buf (IN ta_keyname_t *key_name,
                IN const void *in_key,
                IN size_t len, OUT void ** out_key);

```

Figure 4. Token Interaction Functions

departs or returns. To do so, the application registers a callback: the name of the function to call in the event of departure or return. The transient authentication system polls the user token, and calls each registered callback in the event of a change. To access sensitive resources, the application must first acquire a decryption key. Applications store an encrypted version of the key and then use the token to decrypt it. When the application needs to create a new key, a random encrypted key can be created and “decrypted” using the token.

Unfortunately, many applications assume that authentication need only be checked once or, at best, infrequently. For example, SSH authenticates users only when initiating a remote login; the connection remains in force until explicitly closed. There must also be a mechanism to ensure that applications respond to departure notifications in a timely way; those that do not will be subject to the more drastic step of full address-space protection.

One example application that we have modified is the open-source, Mozilla web browser; we are currently working on extending support to other applications. Mozilla, and web browsers in general, store sensitive information in numerous places. So far we have identified the browser cache, password manager, SSL key store, certificate store, and the cookie store as the most critical. The use of programming language tools to find such sensitive material is an important open problem.

After registration with the API, Mozilla uses the token to generate fresh keys for each of these resources. Although the actual implementation for each of the secrets is slightly different, each remains encrypted when the user is

not present. While the user is present, the browser has the capability to decrypt this information and use it. Otherwise, requests return an error message to the user interface.

5 Related Work

Several efforts have used proximity-based hardware tokens to detect the presence of an authorized user. Landwehr [7] proposes disabling hardware access to the keyboard and mouse when the trusted user is away. This system does not fully defend against physical possession attacks. At the very least, the contents of disk and possibly memory may be inspected at the attackers leisure. Similar systems have reached the commercial world. For example, the XyLoc system [14] could serve as the hardware platform for our authentication token. Our contribution is not the use of a hardware token for proximity detection. Rather, it is the principle of transient authentication and the way in which it affects system and application design.

We have rejected the use of smartcards for authentication services [2]. There are two ways to use smartcards, insertion and swiping. Inserted cards are likely to be left in the machine when the user is away. Swiping must be done frequently, or employ long timeout periods. In either method, smartcards have the same weaknesses as passwords.

One could imagine using biometric authentication rather than a worn, wireless token to provide proximity cues. Unfortunately, biometrics suffer from several usability problems. They have a large false negative rate [12], and are not easily revocable—if someone has a copy of your thumbprint, you cannot easily change it. Furthermore, bio-

metric authentication often requires some conscious action on the part of the user. The one exception is iris recognition [10], but this scheme requires three separate cameras, a bulky and expensive proposition for mobile devices.

There are a number of file systems that provide transparent encryption: Blaze's CFS [1], Zadok's Cryptfs [16], and Microsoft's EFS [8]. None of these tie user authentication to the encryption process properly. Some systems, such as EFS, require the user to reauthenticate after certain events to bound the window of vulnerability. This increases security, but decreases usability.

6 Conclusion

Computing systems currently depend on persistent authentication, in which user authentication is assumed to hold for a long, perhaps unbounded, time. Given the poor physical security afforded by mobile or ubiquitous devices, this is untenable. Instead, we propose the use of transient authentication, in which a user wears a small token that constantly authenticates on his behalf. Transient authentication secures systems against physical possession attacks without compromising performance or usability. We have constructed a cryptographic file system using this approach, and are currently constructing an API to provide support for a broad number of applications and services.

Acknowledgements

The authors wish to thank Peter Chen, who suggested the recovery time metric, and Peter Honeyman, for many valuable conversations about this work. Erez Zadok's FiST framework substantially simplified the construction of ZIA. This work is supported in part by the Intel Corporation; Novell, Inc.; and the Defense Advanced Projects Agency (DARPA) and Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Intel Corporation; Novell, Inc.; the Defense Advanced Research Projects Agency (DARPA); the Air Force Research Laboratory; or the U.S. Government.

References

- [1] M. Blaze. A cryptographic file system for UNIX. In *Proceedings of the 1st ACM Conf. on Computer and Communications Security*, pages 9–16, Fairfax, VA, November 1993.
- [2] M. Blaze. Key management in an encrypting file system. In *Proceedings of the Summer 1994 USENIX Conference*, pages 27–35, Boston, MA, June 1994.
- [3] M. D. Corner and B. D. Noble. Zero-interaction authentication. In *Proceedings of the ACM International Conference on Mobile Computing and Communications*, Atlanta, GA, September 2002. to appear.
- [4] W. Diffie, P. van Oorschot, and M. Wiener. *Design Codes and Cryptography*. Kluwer Academic Publishers, 1992.
- [5] A. Freier, P. Karlton, and P. Kocher. The SSL protocol version 3.0. Internet Draft, March 1996.
- [6] J. S. Heidmann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.
- [7] C. E. Landwehr. Protecting unattended computers without software. In *Proceedings of the 13th Annual Computer Security Applications Conference*, pages 274–283, San Diego, CA, December 1997.
- [8] Microsoft. Encrypting File System for Windows 2000. <http://www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp>.
- [9] C. Narayanaswami and M. T. Raghunath. Application design for a smart watch with a high resolution display. In *Proceedings of the Fourth International Symposium on Wearable Computers*, pages 7–14, Atlanta, GA, October 2000.
- [10] M. Negin, T. A. Chmielewski Jr., M. Salganicoff, T. A. Camus, U. M. Cahn von Seelen, P. L. Venetianer, and G. G. Zhang. An iris biometric system for public and personal use. *IEEE Computer*, 33(2):70–5, February 2000.
- [11] National Institute of Standards and Technology. Computer data authentication. FIPS Publication #113, May 1985.
- [12] P. J. Phillips, A. Martin, C. L. Wilson, and M. Przybocki. An introduction to evaluating biometric systems. *IEEE Computer*, 33(2):56–63, February 2000.
- [13] N. Provos. Encrypting virtual memory. In *Proceedings of the Ninth USENIX Security Symposium*, pages 35–44, Denver, CO, August 2000.
- [14] Ensure Technologies. <http://www.ensuretech.com/>.
- [15] B. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of the First USENIX Workshop of Electronic Commerce*, pages 155–70, New York, NY, July 1995.
- [16] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, 1998.
- [17] E. Zadok and J. Nieh. FiST: a language for stackable file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 55–70, San Diego, CA, June 2000.

Session 2. Operating Systems

1. InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems

Authors: Timothy Roscoe, Richard Mortier, Paul Jardetzky, Steven Hand

page 30

2. Dependable software needs pervasive debugging

Authors: Timothy L. Harris

page 38

InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems

Timothy Roscoe
Intel Research at Berkeley
2150 Shattuck Avenue, Suite 1300
Berkeley, CA, 94704, USA
troscoe@intel-research.net

Richard Mortier
Microsoft Research
7 J.J. Thompson Avenue
Cambridge CB3 0FB, UK
mort@microsoft.com

Paul Jardetzky
Sprint Labs
1 Adrian Court
Burlingame, CA 94010, USA
pjardetzky@sprintlabs.com

Steven Hand
Univ. Cambridge Computer Laboratory
15 J.J. Thompson Avenue
Cambridge CB3 0FD, UK
steven.hand@cl.cam.ac.uk

Abstract

Dependable systems cannot be built without a monitoring and management component. In this paper we propose using a wide variety of information gathering tools coupled with custom scripts and a Prolog language engine to aggregate information from multiple sources. Complex queries, difficult to express in standard database languages, can then be used to answer questions about the system (e.g. the health of individual components) or to discover contradictions (e.g. inconsistent configurations). We describe our prototype implementation and present some early results.

1. Introduction

Today's distributed systems are highly complex dynamic environments where components are strongly dependent on each other for their correct behavior. This situation becomes further complicated as we move toward pervasive and mobile computing since systems must now cope with a huge variety of devices and device software versions. In such systems change is no longer rare: indeed, the state of individual components may be changing rapidly enough that inconsistency and instability is the norm. This may be an unavoidable emergent property of sufficiently large and complex distributed systems (see e.g. recent work on Internet routing stability [7, 9, 14]).

Yet it is crucial that these dynamic and intricate environments be managed: service providers need to sell a de-

pendable service to customers; administrators need to provide a solid infrastructure to users; and users need their ubiquitous networks to be unobtrusively reliable. System management—always a difficult problem—becomes particularly acute when the manager has so little control over so many aspects of the environment, or little idea about how it all fits together [2].

This paper addresses the problem of dependability of complex, dynamic distributed systems. We specifically look at the problem of system health monitoring, answering questions like “Is the system functioning correctly?”, “What is wrong with the system?”, “Is the state of the system in line or at odds with our expectations?”, and “What needs to be fixed to ensure correct functioning of the system?”

We start from an assumption that the system will always admit inconsistency (at least transiently), and from the belief that the goal of dependability can be furthered by capturing and reasoning about these inconsistencies. We use a logic programming language to reason about the state of the system, at least in part since the flexible data representation afforded by such languages is rather more appropriate for dealing with inconsistencies than a rigid database schema.

System information is gathered by using whatever ad hoc or off-the-shelf tools are available; our approach is to build upon existing system and network management tools, not to replace them. Indeed, a key benefit of our approach is the ability to use tools with overlapping areas of applicability, explicitly record the origin of each piece of information gathered, and then reproduce this when resolving contradic-

tions.

In addressing the above problems in this paper, we are explicitly not attempting to provide a mechanism for actively controlling or configuring networks. Neither is InfoSpect intended to provide the kind of rapid response offered by, for example, networking intrusion detection systems. Instead, we focus on providing useful diagnostic information to a management system (which still involves a human being), integrated from a variety of sources, and which can effectively deal with unforeseen and/or contradictory conditions in the system being monitored.

2. Current approaches

Current approaches to determining system state may be split into network-related and host-related schemes. The former tend to be marketed by networking equipment or big-iron vendors and address the traditional FCAPS objectives of network management. Examples include Cisco's CiscoWorks2000, Micromuse's Netcool, Riversoft's OpenRiver, IBM's Tivoli Netview, and the HP OpenView suite.

These systems help operators control their network by providing simplified interfaces to topology discovery, service provisioning and equipment maintenance checks. The Internet community also provides some network-related inspection tools, particularly for checking consistency or syntax of router configuration, for example RPSL at RIPE [13], or the ISI RaToolSet [6].

Commercial host-related systems management software typically focuses on the PC/server oriented enterprise space: e.g. IBM's Tivoli Suite, or Computer Associates' UniCenter. Microsoft's SMS is targetted at smaller systems composed of Windows-based machines.

All these commercial systems, while useful, are insufficient to solve the problems of managing a dependable computing environment since they assume prior knowledge of the correct state of a semi-static set of actors. The diverse, dynamic and ad hoc systems of the future are not well served by such simplifying assumptions.

In the research community, there are efforts to increase system dependability by building operating systems and architectures for distributed and/or ubiquitous computing (e.g. EROS [15], Xenoservers [12], one.world [4], JX [3]). This work is valuable, but we see it as largely orthogonal to our work: no matter how reliable or flexible individual components are, there will always be a need for a distributed monitoring and management function.

To summarize: reliability management based on assumptions of total control within well-defined perimeters starts to look very fragile in the context of dynamically evolving networks of devices and peer-to-peer software systems. To cope with such an environment, management software must itself be ad-hoc and constantly evolving. The re-

mainder of our paper presents our design rationale in further detail, and introduces our prototype implementation and initial results.

3. Our approach: Logic languages

We have codified our motivation for investigating logic languages into a series of (overlapping) design principles which we present below:

Decouple health monitoring from system operation

Most system management tools manage 'before the fact': they tightly integrate the functions of monitoring and control. The emphasis is on deciding on a desired system state, and then making the system elements consistent with that state. While this approach can work well in a highly centralized and controlled environment, complete consistency is an unrealistic goal in a large and complex system. There are several reasons for this:

1. The time taken for the system to converge to the desired state may be comparable with the interval between configuration changes. This has been observed to be the case with many networks and peer-to-peer systems [8, 17].
2. Changes are frequently made independently of the management entity. In many systems, a central management solution simply does not scale socially, since the demands of users for changes to the infrastructure exceed the capacity of the management organization to implement them in a timely manner. As a result, users (whether individuals or organizations) take matters into their own hands. This is not an unusual state of affairs in networking research laboratories, for example. More generally, this is the normal state of affairs in a pervasive computing system.
3. There may be no clearly defined notion of a central management entity anyway, because the system is in constant interaction with others whose configurations are themselves changing. This is again the case for mobile users in a future pervasive computing environment, but is also true for large ISP networks which have peering relationships with other carriers.

An alternative approach to system health monitoring is to manage 'after the fact': construct a view of what the configuration of the system actually *is*, and then allow the operator to manage the system in order to achieve what is desired. This option is more appropriate in the kinds of open and dynamic environments we are interested in here, and our approach falls squarely into this category.

Logic languages such as Prolog [1] seem to have many advantages over the relational databases used in modern system monitoring packages. Prolog is a purely declarative language in which a program consists of *facts* about objects in a system, *inference rules* which define relationships between objects and allow the derivation of new facts, and *queries* which ask questions about objects and their relationships. Facts in Prolog are free-form logical propositions; there is no predefined data schema.

Make it easy to integrate diverse information sources

As well as integrated system management environments, a wide variety of excellent ad hoc system monitoring tools are available both commercially and non-commercially. Such tools work well because they focus on specific functionality – port scanning, SNMP traps, etc. Effective distributed system diagnosis, on the other hand, requires correlating and aggregating information from many sources.

We want to use as many of these sources of information as possible: as systems evolve over time, a system health monitor which can usefully combine the results from other tools will win over an integrated solution which tries to do everything itself.

There are two challenges here: providing a common *representation* of the results from disparate tools so that they can be unified, and the software engineering problem of *interfacing* our monitoring system to these tools.

Prolog performs well in both roles here: in contrast to monolithic software architectures, logic languages are highly effective at unifying the output of other tools via the use of inference rules, and in contrast to RDBM systems, the absence of predefined schema makes it easy to add new information sources.

Expect the unexpected

In large distributed systems, contradictions frequently exist between what the managers believe the system configuration to be and what is independently observed to be the actual system state. These contradictions are usually symptomatic of security problems, faults, or misconfigurations, yet they are unlikely to be detected by monolithic systems based on well defined schema, since the idea of a schema itself always presupposes some consistency of state.

For example, a database which models domain name records as a series of Unix *hostent*-like structures will have no way of representing a situation in which two replica domain name servers have conflicting A-records.

Put simply: relational database systems do not handle contradictions well. The separation between observed system facts and inferences about system state distinguishes InfoSpect from database-oriented monitoring systems.

We note that in this example, as in most others, a relational schema clearly *could* be extended to deal with the situation. Our point is that for unexpected inconsistencies this must be done after the fact, a difficult operation in databases, especially as important information may have already been thrown away. With our approach there is no need to throw away anything. There is no requirement that the set of facts we have be consistent. It is not even required that we have some *a priori* notion of consistency.

Bind assumptions about state as late as possible

The preceding goals and assumptions lead to perhaps our most important design principle: avoid binding any assumptions about the state of the system until the last possible moment. The flexible knowledge representations allowed by declarative logic languages such as Prolog benefit us greatly in this respect.

The same flexibility that allows us to easily integrate diverse tools also allows us to accept data from the tools “without prejudice”, and only later interpret these facts within a particular model of behavior (and check that they are consistent with the model).

This is in contrast to the use of relational databases, where an explicit schema is imposed on observations from the network or system. The process known as “data cleaning” (in data warehousing terminology) when outputs from tools are first fed into the RDBMS has a tendency to throw away precisely the anomalous observations that are most interesting. The use of a schema also makes it hard to evolve the system as new network characteristics become important.

With Prolog, the schema is implicitly present only in the queries and inference rules written alongside the data from network tools, and so can be changed at whim. In fact, we can view the process of making a query as temporarily binding a schema to the data for the duration of the query. This late binding is essential in a highly dynamic and evolving environment.

Of course, the nature of the discovery tools we use also imposes something of an *a priori* schema on the data. Note, however, that this is also the case with systems based on the relational model. Furthermore, by using a diverse array of different discovery tools and delaying the unification of their results until query time, InfoSpect can mitigate the effect of these tool-specific representations of network state.

Finally, we note that the *source* of a particular fact about the system is often as useful as the fact itself in determining state anomalies. In InfoSpect all facts are labeled with where they were obtained. This kind of information is typically discarded in database-oriented systems.

4. Implementation

The InfoSpect prototype consists of a central Prolog knowledge base, a set of *driver scripts*, and a corresponding set of ad hoc or off-the-shelf *discovery tools*.

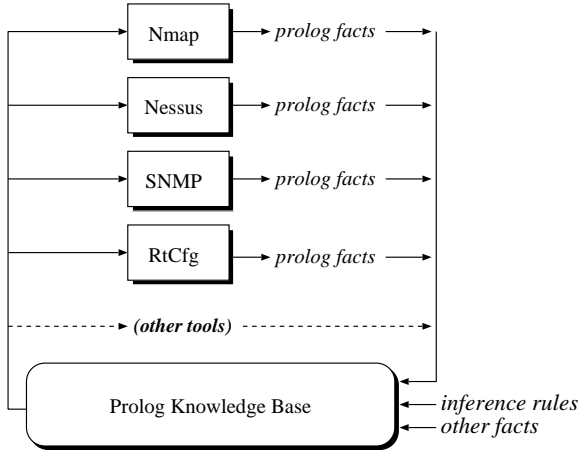


Figure 1. System Data Flow in InfoSpect

Figure 1 illustrates the flow of information through the system. The “main loop” of the system queries the knowledge base for a list of driver scripts to be run, and then invokes each of these (preferably in parallel) to gather information. A driver script queries the knowledge base for information (e.g. the set of possible routers) which it uses to drive a discovery tool. The driver script translates the output of the discovery tool into Prolog facts which are then added to the knowledge base.

This system is highly extensible since it may be updated to use new tools without interrupting normal operation by adding to the knowledge base a set of Prolog facts that describe the set of discovery scripts to be run. These new or modified tools will be run in subsequent iterations.

Entries in the knowledge base come in three flavors:

1. Facts input from external sources (such as a list of well-known Trojan horse ports, or vendor tags for Ethernet MAC addresses);
2. Facts directly observed from some discovery tool (which themselves retain information as to which tool they originate from); and
3. Inference rules which are used to derive new facts from existing ones.

For example, entries of the first kind might include the fact that port 53 is reserved for DNS traffic, entries of the second kind could include which hosts were observed by Nmap to be listening on TCP port 53, and entries of the

third kind could include the idea that a machine listening on port 53 is likely to be a DNS server.

4.1. Example tools

Some concrete examples of driver scripts and discovery tools used in our current implementation are:

Network discovery tools: We have a simple SNMP walker written in Python which writes topology and routing information to the knowledge base, and a network consistency checker which queries the knowledge base for likely routers and fetches the running configurations from those routers. The knowledge base includes the router’s bootstrap configurations and so the checker can determine what, if anything, has changed.

Our experience has been that automatically interpreting router configurations obtained in the traditional way using expect scripts is easier if the information is converted into Prolog as early as possible in the process, and the rest of the job implemented as Prolog rules.

Using this information, predicates can be written to (for example) dynamically check for consistent BGP filters and policies. Another application is determining whether it is possible to transmit an IP packet out of an intranet without traversing a firewall box—a useful feature in a network testing lab.

Host system scanners: Several driver scripts are used for discovering hosts and checking host system security, including the network mapper Nmap [5], and the remote security scanner Nessus [16], fed with the facts derived from the network discovery tools. The results are facts like:

```

nmap_ipaddr('10.64.201.201').
nmap_ipaddr('10.64.201.209').
...
nmap_os('10.64.201.201','Solaris 2.6 -
2.7').
nmap_os('10.64.201.209','Foundry Server-
Iron XL Switch Version 06.0.00T12').
...
nmap_tcp_port_open('10.64.201.201',22).

```

—which indicates among other things that, according to Nmap, a host with IP address 10.64.201.201 exists, and is listening on the ssh port, and probably runs Solaris¹.

From these results, predicates can be derived to locate security anomalies in a variety of end systems. A simple example might be to ask the system for all Windows machines running a vulnerable version of IIS.

These facts are used by the system in other ways as well. A heuristic for discovering routers incorporates operating system information and would include the machine

¹Nmap includes an operating system fingerprint capability, used here.

10.64.201.209 above as a result. We will see in the next section how heuristics like this can be very concisely expressed.

4.2. A detailed example: DNS walker

In this section we present a much more detailed discussion of one example driver script, to give a more concrete feel for how the system works.

The purpose of the DNS walker is to acquire as many DNS records as possible from as many DNS servers as possible in the local network, and add this information to the knowledge base. As well as requesting zone transfers from DNS servers, the walker repeatedly and recursively makes forward and reverse name lookups for all the host names and addresses it can find. The walker is fairly straightforward, and written in Python; we concentrate here on the Prolog operations related to it.

The walker starts by requesting all values which match the predicate:

```
likely_dns_server(X).
```

This predicate is a pre-defined heuristic for spotting DNS servers, it's simply defined as:

```
likely_dns_server(Machine) :-
    server(Machine, domain).
```

The `server` predicate is a similarly-defined heuristic for spotting servers in general; it's a little more interesting:

```
server(Machine, Svc) :-
    ipservice(Svc, Port, tcp, _),
    nmap_tcp_port_open(Machine, Port).
server(Machine, Svc) :-
    nmap_tcp_port_open(Machine, Svc).
```

This allows us to specify services by name (as in the example above) or port number. The upshot if this is the system regards a machine as a *potential* DNS server if Nmap saw it listening on the domain port.

The level of indirection afforded by the `likely_dns_server` heuristic is important. It gives us a way to flexibly integrate tools without introducing fragile dependencies between them: the DNS walker does not need to be aware of the operating of Nmap, and we could remove Nmap and substitute some other source of port information if we wanted, or indeed do without automatic discovery of DNS servers altogether, and manage with a set of user assertions about which machines should be queried for DNS records.

Conversely, the wrapper for Nmap does not need to concern itself with outputting facts in a form friendly to the DNS walker. The Prolog functions we have listed above give a flavor of the conciseness and flexibility in tool integration that the use of a logic language affords us.

For each potential DNS server, the walker starts from an initial list of host IP addresses, obtained in a similar manner from the results of running previous tools, and performs its walk over the record space of the server. The output of the walker consists of two types of Prolog facts. The first simply confirms that a DNS server was found at a particular address; it's therefore a stronger statement than `likely_dns_server(X)`. For example:

```
dns_working('10.64.201.201').
dns_working('10.64.209.2').
...
```

The second type of fact indicates the existence of a DNS record of a particular type, in a particular server, with particular name and value. For example:

```
dns_record('10.64.201.201', 'A',
    'kristeva.smoke.sprintlabs.com.',
    '10.64.202.54').
```

—indicates that the DNS server 10.64.201.201 has a A record specifying that `kristeva.smoke.sprintlabs.com` has IP address 10.64.202.54. All information is kept, including the address of the server which supplied the record. Even the name of the predicate indicates that the DNS walker, and not some other driver script, was the source of the information.

These facts are now available as input to other discovery tools (for example, a driver script might require a list of machines pointed to by MX records), and can also be queried for inconsistent or anomalous configurations. For instance, the following Prolog function will uncover servers with inconsistent A-records:

```
dns_has_two_arecs(Server, V, N1, N2) :-
    dns_record(Server, 'A', N1, V),
    dns_record(Server, 'A', N2, V),
    N1 @> N2.
```

More sophisticated queries are also possible. For instance, this Prolog function succeeds if a DNS server has an inconsistent pair of PTR and A records (i.e., forward and reverse address/name bindings):

```
dns_ptr_conflict(Server, Ptr, DNS, A) :-
    dns_record(Server, 'A', DNS, A),
    dns_record(Server, 'PTR', Ptr, DNS),
    not(ip_arpa_equiv(A, Ptr)).
```

The `ip_arpa_equiv(A, Ptr)` function succeeds if `Ptr` is the “.inaddr.arpa” representation of `A` (or vice versa).

4.3. Performance

We ran InfoSpect in the Sprint Labs internal network of about 300 hosts, more than 20 of which are IP routers. With

our current suite of discovery tools, this results in a knowledge base of around 25,000 entries.

There are two aspects to the performance of the system: time taken to perform a query against the knowledge base, and time taken to collect information from the network. Together these determine how up-to-date the information in the knowledge base can be, and how quickly this information can be interpreted to diagnose problems.

Query performance is good: a typical query (for example, to determine routing table inconsistencies) takes only a few milliseconds to execute on a modern workstation. We have not addressed the issue of formulating queries so as to optimize performance at the Prolog level since this has not been a problem so far. Recent work on high-performance declarative languages such as Mercury [10] may help to address this below the level of the language.

Runtime of discovery tools, on the other hand, is dominated by network communication latencies and, more significantly, by the need to throttle some network probes to prevent undue load on the systems being monitored. Many of our driver scripts (such as the port scanners) take on the order of minutes to complete. Consequently, the knowledge base is always a few minutes behind the state of the system. On this basis, InfoSpect does not fall into the “real-time anomaly detection” class of applications, but the speed of queries does allow complex, unanticipated questions to be asked and answered in a timely fashion.

5. Ongoing work

There is much short-term work left to do in InfoSpect. The addition of more discovery tools to the collection we have now will extend the scope of the system but also allow us to gain more experience with constructing the Prolog functions that integrate tools and interpret the knowledge base.

A larger unresolved issue is how best to represent time in InfoSpect: the knowledge base at present holds only observations about the network from the loosely-defined “recent past”. A natural first step is to add a timestamp to every fact in the knowledge base which is directly generated from a discovery tool – note that this change can in fact be made to a running InfoSpect system by adding a trivial rule for every class of observation which removes the timestamp. Inference rules and queries could then be formulated over a set of similar facts with different timestamps, but we would prefer a more structured approach to the problem. Two promising avenues to explore are the kind of timing techniques employed in high-level hardware design languages, and temporal logics. A challenge will be to implement a framework for handling time without unduly expanding the state space of facts we have to deal with.

While we are unaware of other work using logic languages to integrate networking monitoring tools, the use of rule- and/or event-based systems in network monitoring is well established. A recent development has been to apply techniques from data mining and machine learning to derive a set of empirical rules about the “normal” behavior of a distributed system or network, and use these in turn to detect anomalies [11]. Such work is complementary to ours and operates at a higher level of abstraction, we speculate that InfoSpect’s knowledge base offers a rich foundation on which to build such tools.

6. Conclusion

Network management and configuration is an increasingly important and complex part of any corporation’s business. Logic languages appear to offer significant advantages in simplifying the difficult tasks of network and security administration. We have built a prototype system which has been used to monitor the local intranet, and in doing so uncovered hitherto unknown inconsistencies and potential security vulnerabilities.

Our experience so far has been good: wrapping existing network tools so that they are both driven from the contents of the knowledge base and deposit their results into the knowledge base has proved relatively simple. By decoupling our approach from the infrastructure as much as possible, we avoid jeopardizing the dependability of the system we are trying to manage.

Prolog has proved to be highly effective at unifying the results of disparate tools. Furthermore, Prolog queries to uncover inconsistencies in the network state (for example, duplicate or potentially forged DNS records) are remarkably concise and intuitive; typically three or four lines.

We are currently extending the work on router and routing policy configurations to provide the kinds of answers network operators need in the daily running of complex networks and services, including backbone networks.

7. Acknowledgments

We would like to thank John Larson and the Sprint Labs Infrastructure Group for letting us loose on the Lab network in the early stages of this work. We are also grateful to the anonymous reviewers of this paper for several insights and useful suggestions.

References

- [1] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer Verlag, 1984.

- [2] P. Dourish, D. Swinehart, and M. Theimer. The Doctor Is In: Helping End Users Understand the Health of Distributed Systems. In *Proceedings of the IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, December 2000.
- [3] M. Golm and J. Kleinoeder. Ubiquitous Computing and the Need for a New Operating System Architecture. Online at <http://www4.informatik.uni-erlangen.de/Projects/JX/Papers/ubitools01.pdf>, 2001.
- [4] R. Grimm, T. Anderson, B. Bershad, and D. Wetherall. A System Architecture for Pervasive Computing. In *Proceedings of the 9th ACM SIGOPS European Workshop, Kolding, Denmark*, pages 177–182, September 2000.
- [5] Insecure.org. The Nmap stealth port scanner. <http://insecure.org/nmap/>, 2001.
- [6] ISI. RAToolSet. <http://www.isi.edu/ra/RAToolSet/>, 2001.
- [7] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed Internet Routing Convergence. In *Proceedings of ACM SIGCOMM 2000*, pages 175–187, 2000.
- [8] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Observations on the dynamic evolution of peer-to-peer networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, USA, March 2002.
- [9] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfiguration. In *Proceedings of ACM SIGCOMM 2002*, August 2002.
- [10] T. Mercury Project. <http://www.cs.mu.oz.au/research/mercury/>, 1996.
- [11] M. N. nez, R. Morales, and F. Triguero. Automatic discovery of rules for predicting network management events. *IEEE Journal on Selected Areas in Communications*, 20(4):736–745, May 2002.
- [12] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: Accounted Execution of Untrusted Code. In *Proceedings of the fifth Workshop on Hot Topics in Operating Systems (HotOS-VII)*, 1999.
- [13] RIPE. RPSL. <http://www.ripe.net/ripenncc/public-services/db/irrtoolset/documentation/>, 2000.
- [14] A. Shaikh, L. Kalampoukas, R. Dube, and A. Varma. Routing Stability in Congested Networks: Experimentation and Analysis. In *Proceedings of ACM SIGCOMM 2000*, pages 163–174, 2000.
- [15] J. S. Shapiro, S. J. Muir, J. M. Smith, and D. J. Farber. Operating System Support for Active Networks. Technical Report MS-CIS-97-03, University of Pennsylvania, February 1997.
- [16] The Nessus Project. <http://www.nessus.org/intro.html>, 2000.
- [17] B. Wilcox-O’Hearn. Experiences deploying a large-scale emergent network. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, USA, March 2002.

Dependable Software Needs Pervasive Debugging

Timothy L. Harris

University of Cambridge Computer Laboratory
J J Thomson Avenue, Cambridge, UK
tim.harris@cl.cam.ac.uk

Abstract

Nobody would claim that debugging computer software is easy: all too often it proceeds by trial-and-error experiments in which programmers examine the behaviour of the system and form hypotheses that could explain what they see. These problems are exacerbated when developing distributed, peer-to-peer or multi-processor applications, or when unreliable network links form part of the system under test. Environments for pervasive computing take this to an extreme, allowing user-supplied code to run or migrate within and around the network.

In this paper we show how to perform pervasive debugging, enabling complex multi-process applications to be debugged and controlled as single entities and their robustness to changes in network performance to be evaluated. We do this by virtualizing the resources used by the system, allowing the threads that it involves and the network links that it uses to be modelled within a single controllable process.

1 Introduction

Debugging computer programs often proceeds by intuition: given some reports of how the system fails the programmer must try to deduce what might have led to those failures and then re-run the program in the hope of reproducing the same problem, or at least gaining a certain amount of corroborative or conflicting information. Eventually sufficient evidence may be gathered to focus the programmer's attention on the root cause of the problem.

This debugging process occurs even in simple projects – where the program is being run within a debugging tool, where it operates on a single computer or where it performs no external communication. This is lamentable: the system is deterministic and running in isolation, there should be no need to run and re-run it experimentally.

These problems multiply with the emergence of *pervasive computing* environments. For instance, in the

Xenoserver project we are building an infrastructure for wide-area distributed computing [15]. We envisage a world in which Xenoserver platforms are scattered across the globe and are available for any member of the public to submit code. Grid computing and programmable networks present similar acute challenges [17]. Current methods for debugging distributed applications require the user to orchestrate separate tools attached to each of the processes involved; a tedious manual task.

In the remainder of this paper we first identify four key problems (P1-P4) faced by users of current debuggers and we then introduce the technique of *virtualized debugging* as a solution to these problems.

2 Challenges in Debugging

We believe that there are four particular problems that exist with conventional debugging tools:

P1. Stop/Inspect/Go Interface. The abstraction presented by a traditional debugger is of a controllable processor with support for stepping through the code, for setting breakpoints at which execution should halt and for inspecting (perhaps updating) the contents of memory locations in terms of the variables manipulated by the source code. However, the programmer must typically either set a breakpoint before a problem develops and step forwards (a slow process if the problem is erratic or its origin uncertain), or set a breakpoint when a problem is detected and examine the system's state to try to deduce how it reached that point. Such archaeology wastes programmers' time.

One solution is to generate extensive logging for off-line analysis. Larus' *whole program paths* [10], and the extensions proposed by Zhang and Gupta [19] represent the state of the art, recording the complete control-flow history of a single thread in a reasonably compact form. An alternative combines logging with *re-execution*, typically recording information during 'forwards' execution and using this to recover intermediate states during 'reverse' execution. The main problems are how to manage these logs

efficiently, how to handle I/O and how to expose a natural *step backwards* operation to the user. In single-threaded systems Boothe's recent paper presents good approaches to all of these problems and extensive references to previous work [2].

P2. Risk of Masking Bugs. In shared-memory multiprocessor systems it is typical that different threads do not see memory access operations in a consistent total order – for example if one thread writes to location *A* and then to location *B* then a second thread may well read the new value of *B* and subsequently the old value of *A*. Such orderings might be caused by caching or write buffering within the processors, or by more advanced techniques such as value-based speculation. Adve and Gharachorloo provide a survey and tutorial of the subject [1]. The same problem exists within multi-threaded virtual machines, either through runtime code optimization or directly through the underlying processor [14].

Practical mechanisms for identifying bugs caused by these low-level problems have not progressed beyond asking experienced programmers to inspect the code. This straightforward approach can work well in a collaborative environment [7], but it does require an established and cooperative community of experts. Moreover, the uptake of SMP and SMT systems will accentuate the need for tools that support fine-grained concurrency, both within application code and within the OS.

P3. Poor Support for Concurrency. Co-operation is required but often lacking between the debugger and the thread system's implementation. For example the debugger may be aware of the threads managed by the operating system kernel, but be unaware of how application threads are multiplexed over each of these system threads (a problem we encountered using dbx under Tru64 UNIX). Conditional breakpoints must still be set in terms of the state of individual threads – not system-wide properties such as 'stop if Thread 1 holds lock A and Thread 2 holds lock B'. More generally operations such as single-stepping or resuming one thread do not take into account other threads in the system – for example whether they run freely, step forwards a similar amount or are suspended.

Deterministic replay schemes have been designed to allow consistent re-execution of multi-threaded processes. LeBlanc and Mellor-Crummey developed a system *Instant Replay* which, during forwards execution, logs the relative order of significant events by associating a version number with each shared object and tracking which threads access which versions [11]. For efficiency they assume that shared state is governed by multiple-reader single-writer locks. Choi and Srinivasan's work is typical of more general schemes in which significant events include accesses to

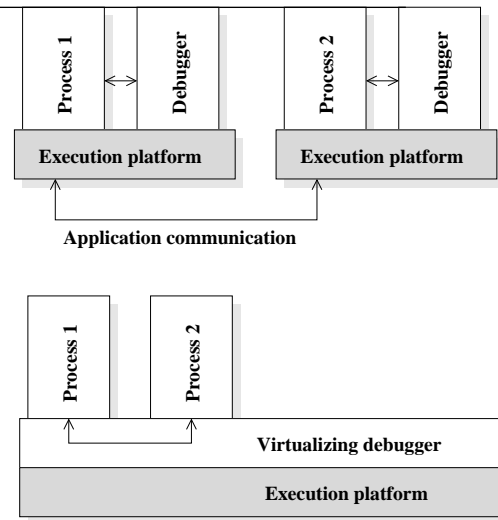


Figure 1. A traditional system (top) and a virtualizing debugger (bottom). Both are shown hosting a two-process distributed system. In the first case two separate machines and debuggers must be used; in the second case a single debugger provides two virtualized environments and communication occurs under its control.

shared memory and operations on mutual exclusion locks and condition variables [5].

A complementary approach is to identify classes of error automatically. The *Eraser* system is a dynamic data race detector [16] using binary rewriting to update per-location records of the set of locks protecting that address in the heap. Each time a location is accessed the associated lock set is intersected with the set of locks currently held: a warning is reported if the lock set becomes empty.

All these systems assume an interleaved model of single-process execution and so do not consider P2 or extensions to distributed systems.

P4. Poor Support for Distribution or Communication.

Distributed programs pose even more substantial problems than those of multi-threaded or multi-processor systems: separate debuggers must be attached to the various processes involved. There is no central way to control system-wide parameters – e.g. to impose loss patterns or delays on communications, to corrupt or duplicate messages, to insert spurious ones or to control the relative execution speeds of threads. Such features must either be intrinsic in the platform running the tests or must be implemented as additional testing code by the programmer.

The Pilgrim debugger provides support for debugging distributed systems built using remote procedure call (RPC) [6], allowing call histories to be shown across machine boundaries. The *p2d2* distributed debugger provides a unified user interface to processes on different machines, and allows for interaction with e.g. MPI libraries [9]. We are not aware of any systems that attempt to support more general communication, for example at the level of TCP, UDP or indeed ‘raw’ sockets.

3 Virtualized Debugging

Each of the challenges identified in Section 2 arises because aspects of the system’s behaviour depend on functions implemented outside the debugger’s control. To address these problems we propose the technique of *virtualized debugging* in which all of the resources used by the system under test are virtualized by the debugger – ranging from low-level details such as the precise implementation of processor instructions to the scheduling of threads, the provision of separate virtual address spaces to different processes and network communication between those processes. Retaining control over the resources in use allows the debugger to ensure deterministic execution, to expand or contract the detail with which parts of the system are modelled, to manage distributed applications as single entities and to control the performance of external components such as communication links.

Conceptually, as shown in Figure 1, this places a single debugger below the entire system that it is being used to study, rather than having separate debuggers attached to each process. Brewer and Weihl suggested a similar structure for debugging high-performance parallel applications on a workstation-hosted simulator – we take their approach to an extreme by considering all resources and multiple machines [3]. In contrast to their processing-based environment, the execution of many distributed applications is dominated by communication latencies: programs being debugged may sometimes actually run faster under the debugger.

3.1 Scope

We envisage virtualized debugging being most useful in developing applications that use a moderate number of communicating threads or communicating processes – perhaps up to a dozen – each operating using the same instruction set and linked against the same libraries. Of course, allowing more heterogeneity would allow the system to have even broader applicability; but we see no shortage of problems to tackle given our assumed environment (either as constructors of the debugger, or as its eventual users).

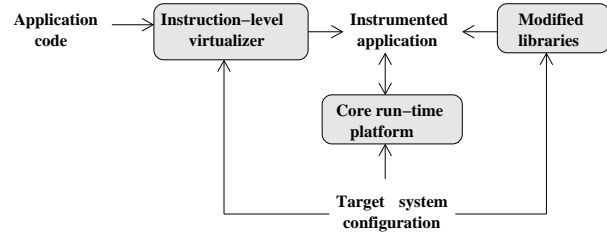


Figure 2. The interaction between the system components.

Hosting the entire system within a single debugger obviously imposes practical limits since it must be executed on a single machine. Raising these limits provides an avenue for future work, perhaps by re-distributing the virtualized environment. However, during our own work on lock-free data structures we have found diminishing returns, in terms of the number of bugs observed, going beyond four concurrent threads.

An interesting question is the extent that virtualized debugging can be used on code forming part of an OS. This is another ‘classical’ environment in which a lack of reproducible behaviour hinders methodical development – for instance the occurrence of a deadlock may depend on the locks held by the application executing on the CPU to which an interrupt is delivered. Our current design will not support such an environment directly: although virtualized kernel-mode execution has been attempted elsewhere many times (for example recently by VMWare) incorporating it here raises a number of additional challenges. The first is the technical difficulty of virtualizing access to devices at the lowest level – that is, through I/O operations, DMA and the like. Secondly, the virtualizing debugger would require knowledge of the particular notions of ‘thread’ and ‘process’ that are used by the operating system it is hosting. We view the most direct application of virtualized debugging to OS implementation as being through the development of reliable libraries and modules from which the OS is constructed.

3.2 System Structure

The structure of the virtualizing debugger can be divided into three components: the *instruction-level virtualizer*, the *modified libraries* and the *core run-time platform*. The interaction of these is shown in Figure 2.

The *instruction-level virtualizer* is responsible for transforming the application code to allow it to be executed directly during debugging. For example by:

- Inserting periodic *yield* operations to return control to the main loop of the debugger (e.g. by using a software instruction counter [13]);
- Using sandboxing to check that memory accesses made by the application are to valid addresses [18]. This is necessary to prevent it from interfering with the enclosing debugger;
- Expanding accesses to shared memory locations into code sequences that emulate access-reordering, write-buffering or caching;
- Giving processes in a multi-process application the illusion of access to separate virtual address spaces, either by integrating operating system support or, more generally, by adding a per-process offset to each memory access made.

This transformation may either be performed ahead of time, or actually be performed dynamically on program ‘hot spots’ with a simple emulator used for uncommon code. External calls made by the resulting *instrumented application* are resolved against *modified libraries* – for example to emulate network access for communication between processes (perhaps introducing loss or delay). The modified libraries can also record the behaviour of I/O operations if reverse execution is to be supported. The *core run-time platform* is responsible for controlling the execution of the threads. During forwards execution, its main loop selects which thread should continue next and then executes that thread until it yields control.

Thread Scheduling. The implementation of the system requires care in order to avoid it masking further classes of bug. In particular, a simple regular placement of yield operations within the instrumented application risks limiting the execution schedules that could occur. Of course, the debugger could systematically explore all possible execution schedules around a program point; there is a clear trade-off between the number of schedules tested and the eventual coverage. However unlike traditional debugging the range of schedules could be changed dynamically, allowing the user to focus on specific parts of the program. In contrast, existing tools either require complete re-execution for each schedule, or perform exhaustive testing over entire runs (an impractical solution for non-trivial software) [4].

An alternative option is to ensure that the thread scheduling implemented by the virtualizing debugger remains typical of the behaviour that the real scheduler would provide. This could be achieved by having the run-time platform dynamically enable and disable a larger set of yield points so that, over a long execution run, thread switches would be considered at each possible location. If thread switches are

based on a software instruction counter then the counter values that trigger switches could be drawn from a random distribution. A complementary approach would be to introduce a random (virtual) delay upon each thread switch to reduce the weak coupling effects that may lead to threads moving in lock-step [8].

Binary Re-writing. The concern of identifying access to ‘external’ resources is, of course, superficially similar to previous work on executing untrusted binary code – Wahbe *et al* introduce that area in their work on Software Fault Isolation (SFI) [18]. However, here we can benefit from closer integration between the debugger and the remainder of the tool chain – e.g. by introducing sandboxing checks before optimization rather than using simplistic binary rewriting, or by exploiting guarantees made by the language’s type system.

Efficient Execution. The structure of distributed applications can be exploited to aid efficient execution over a virtualizing debugger: for instance, different nodes within a peer-to-peer system may run concurrently where allowed by the communications that they attempt. Similarly, if the instrumented application supports roll-back, then the core run-time platform may execute multiple threads from within the same process, check whether they did make conflicting memory accesses and, if they did, step back to before the conflict and then re-run sequentially. In each case the user is presented with the illusion of a single deterministic system whose execution they can control at all levels.

3.3 Dependable Systems

The availability of efficacious debugging tools does not in itself automatically lead to dependable computing systems. However, aside from the general practical benefits that virtualized debugging can bring to the software development cycle, there are extensions that could be of particular use for dependability. In particular, by exposing instrumentation and control interfaces for thread scheduling, it would be possible to express a variety of alternative tools using this same framework.

Firstly, simple coverage testing can be performed by recording each range of addresses that are executed and subsequently identifying code sections that have not been exercised. Secondly, for smaller programs, exhaustive testing is possible. This could be performed directly at a low level, testing each thread schedule in turn up to a specified depth or until a previously-observed state is reached. Alternatively, the programmer could define a mapping function from the concrete state of the system to a logical state and the search could explore that logical state space.

An important attraction of both techniques over symbolic model checking is that they operate using the same code that will ultimately be executed. While any exhaustive technique has limited scalability, integrating state-space exploration with the debugger could allow the programmer to initially run threads to a point at which problematic behaviour is observed and then use exhaustive exploration around that point to search for problems. Effectively this allows a limited test case to be generated directly from the full system, rather than requiring that it be extracted by hand.

4 Conclusion

Virtualized debugging provides an effective solution to the four problems identified in Section 2:

- P1 Arbitrary system states can be recovered by re-running the system from a previously recorded position. As with Boothe's work we can trade-off between the frequency with which checkpoints are taken and the time required to recover an intermediate point.
- P2 Arbitrary levels of processor detail can be included by the appropriate expansion of instructions. In addition to modelling memory accesses we could consider, for example, cache behaviours, TLB performance or the availability of specialized functional units.
- P3 The debugging interface has control over the scheduling policy and can therefore provide deterministic re-execution and selective stepping (or reversing) of specific threads.
- P4 By supporting multiple virtual address spaces within a single debugger it is possible to use the same framework with distributed applications. Breakpoints can be set according to system-wide properties and emulated communication links can be made subject to spurious transmission, to loss or to duplication.

In this paper we have outlined the technical infrastructure needed to support virtualized debugging: a further challenge is how to expose the facilities of this system to programmers. For example, in terms of the interfaces provided to control execution, or to set the delay and loss characteristics of links, or the thread and process scheduling policies.

Our implementation work is ongoing, building on techniques we developed for a currently-unpublished tool used to allow shared libraries containing static data to be used with our single-address-space Nemesis operating system [12] and on the open-source binary-instrumentation Valgrind tool. We hope to demonstrate a prototype at the Workshop in September 2002.

The delivery of dependable computer systems must be underpinned by the testing and evaluation of each of the

components involved. As we have shown, existing debugging tools provide few of the facilities desired by today's programmers, let alone tomorrow's. Virtualized debugging provides a key remedy for this, applicable to a spectrum of settings ranging from distributed, peer-to-peer and agent based applications down to the implementation of concurrency primitives within a multi-processor OS.

References

- [1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.
- [2] B. Boothe. Efficient algorithms for bidirectional debugging. In *Programming Language Design and Implementation (PLDI '00)*, volume 35(5) of *ACM SIGPLAN Notices*, pages 299–310, May 2000.
- [3] E. A. Brewer and W. E. Weihl. Developing parallel applications using high-performance simulation. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 158–168, May 1993.
- [4] D. Bruening. Systematic testing of multithreaded Java programs, 1999.
- [5] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, Aug. 1998.
- [6] R. C. B. Cooper. *Debugging concurrent and distributed programs*. PhD thesis, University of Cambridge Computer Laboratory, Feb. 1988. Also available as UCAM-CL-TR-128.
- [7] J. Domingue and P. Mulholland. Fostering debugging communities on the Web. *Communications of the ACM*, 40(4):65–71, Apr. 1997.
- [8] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *ACM Transactions on Networking*, 2(2):122–136, Apr. 1994.
- [9] R. Hood. The *p2d2* Project: Building a Portable Distributed Debugger. In *Proceedings of ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'96)*, Philadelphia, PA, May 1996.
- [10] J. R. Larus. Whole program paths. In *Programming Language Design and Implementation (PLDI '99)*, volume 34(5) of *ACM SIGPLAN Notices*, pages 259–269, May 1999.
- [11] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr. 1987.
- [12] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, Sept. 1996.
- [13] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '89)*, Apr. 1989. ACM SIGARCH Computer Architecture News 17(2):78–86, April 1989.

- [14] W. Pugh. Fixing the Java memory model. *Proceedings of the ACM 1999 Conference on Java Grande*, pages 89–98, June 1999.
- [15] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: accounted execution of untrusted code. In *Proceedings of the fifth Workshop on Hot Topics in Operating Systems (HotOS-VII)*, 1999.
- [16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.
- [17] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.
- [18] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of 14th ACM SOSP*, pages 175–188, Dec. 1993.
- [19] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *Programming Language Design and Implementation (PLDI '01)*, volume 36(5) of *ACM SIGPLAN Notices*, pages 180–190, May 2001.

Session 3. Theory

1. Specifying and Verifying Systems With TLA+
Authors: Leslie Lamport, John Matthews, Mark Tuttle, Yuan Yu
page 44
2. Rigour is good for you and feasible: reflection on formal treatments of C and UDP sockets
Authors: Michael Norrish, Peter Sewell, Keith Wansbrough
page 49
3. Capturing OS Expertise in a Modular Type System: the Bossa Experience
Authors: Julia L. Lawall, Gilles Muller, Luciano Porto Barreto
page 54

Specifying and Verifying Systems With TLA^+

Leslie Lamport
Microsoft Research

John Matthews
HP Labs
Cambridge Research Lab
Cambridge, MA

Mark Tuttle
HP Labs
Cambridge Research Lab
Cambridge, MA

Yuan Yu
Microsoft Research

Abstract

TLA^+ is a high-level specification language that has been used to specify and check the correctness of several hardware protocols. We expect that it can also be used to specify and check concurrent algorithms and protocols for software systems.

1. Introduction

Correct code is an important component of software reliability. Modern operating systems make extensive use of concurrent and distributed algorithms. These algorithms are subtle and easy to get wrong, leading to an incorrect high-level design. We are concerned with formally describing high-level system designs and checking their correctness. We also address the question of checking that code implements the high-level design.

Systems should be described in a formal specification language to produce unambiguous descriptions that can be checked with tools. Conventional programming languages are ill-suited to this task because: (i) they describe one way of doing something, while a high-level description should allow many implementations; (ii) the need to generate efficient code makes them complicated; and (iii) they do not provide the mathematical abstractions required for simple, high-level specifications. We use the language TLA^+ . We believe that its simplicity, elegance, and expressiveness make it ideal for writing high-level descriptions of concurrent and distributed systems.

We have had considerable experience applying TLA^+ to hardware protocols. Our experience suggests that it should be good for software as well. Section 2 describes TLA^+ and its associated tools and techniques; section 3 describes our experience using it; and section 4 discusses its potential application to software systems.

2. TLA^+ and Friends

2.1. TLA^+

TLA^+ is a formal specification language based on (untyped) ZF set theory, first-order logic, and TLA (the Temporal Logic of Actions) [4, 7]. TLA is a temporal logic developed for describing and reasoning about concurrent and distributed systems [5]. TLA^+

includes modules and ways of combining them to form larger specifications.

Although TLA^+ permits a wide variety of specification styles, a typical specification has the form $Init \wedge \Box Next \wedge Liveness$, where:¹

Init is the initial-state predicate—a formula describing all legal initial states.

Next is the next-state relation, which specifies all possible steps (pairs of successive states) in a behavior of the system. It is a disjunction of actions that describe the different system operations. An action is a mathematical formula in which unprimed variables refer to the first state of a step and primed variables refer to its second state.

Liveness is a temporal formula that specifies the liveness (progress) properties of the system as the conjunction of fairness conditions on actions. Although the specifications we wrote included liveness properties, these properties were not checked during the verification, so we largely ignore liveness here.

Such a specification essentially describes a state machine. However, unlike specifications in methods based on abstract machines or programming languages, the state predicates and actions in TLA^+ specifications are arbitrary formulas written in a high-level language with the full power of set theory and predicate logic, making TLA^+ very expressive. A TLA^+ specification consists of a single mathematical formula.

2.2. TLC

TLC [11] is an on-the-fly model checker for debugging TLA^+ specifications. This distinguishes it from almost all other model checkers, which require specifications to be written in primitive, low-level languages. No model checker can handle all the specifications that can be written in a language as expressive as TLA^+ . However, TLC can handle a subclass of TLA^+ specifications that seems to include the ones that arise in describing actual systems.

Explicit-state model checkers like TLC must generate all reachable states. Real system specifications are usually not finite state—for example, they may contain unspecified sets of processors and

¹The formula $\Box Next$ should actually be $\Box [Next]_v$, where v is the tuple of all variables; we drop such subscripts for simplicity. We also ignore the hiding of internal variables, expressed with temporal existential quantification.

unbounded message queues. We run TLC on the actual specification, using a separate configuration file that specifies a finite-state instance. TLC can also be used to generate finite-length random simulations of even infinite-state specifications.

TLC can be used to check safety and liveness properties of a specification written in the form $Init \wedge \Box Next \wedge Liveness$. For safety properties, TLC explores all reachable states in the model, looking for one in which an invariant is not satisfied or deadlock occurs (there is no possible next state). For liveness properties, TLC uses the standard tableau method described in [9]. TLC avoids the tableau construction of the temporal formula if it contains only temporal subformulas of the form $\Box \Diamond A$ or $\Diamond \Box A$, where A is a state or action predicate. This allows it to handle SF (strong fairness) and WF (weak fairness) properties more efficiently. Because the state graph constructed for liveness checking can easily be too large for TLC to handle, TLC periodically checks the liveness properties on the partial state graph constructed so far. When TLC detects an error, a state trace that exhibits the error is printed as part of the error report. For a violation of a safety property, the error trace is guaranteed to be minimal-length.

There are a number of interesting features that makes TLC a useful tool.

- TLC keeps all of its internal data structures for invariant checking on disk, using main memory as a cache to provide efficient implementation of those data structures. This allows us to explore large state spaces—eliminating the common way in which explicit-state model checkers run out of memory. (The largest state space TLC has checked so far contains about 900 million distinct reachable states.)
- TLC can run in multi-threaded mode to make use of multi-processors, and in distributed mode to make use of multiple machines. It obtains a speedup that is nearly linear in the number of processors. TLC users routinely check their specifications using large multiprocessor systems and/or multiple machines.
- TLC periodically generates checkpoints during its runs. A checkpoint can be used to resume an old run after a system crash, or even after correcting a minor error found by TLC. Because its checkpointing is very fast, TLC can afford to take checkpoints fairly frequently during a long run. TLC users have found this feature quite useful.
- TLC allows a user to specify symmetries in a specification by providing a group of symmetry-preserving permutations. It then applies the standard symmetry-reduction techniques to explore only the quotient state space modulo the permutation group.

Because TLA⁺ is such a high-level language, we expect TLC to be slower than comparable model checkers—perhaps by a factor of ten in the single-threaded mode, although we have not made any real effort to compare it with other systems. But TLC can make use of multiprocessors and multiple machines when better performance is needed. Symbolic model checkers outperform explicit-state ones on many problems. A new TLA⁺ model checker based on satisfiability solving is therefore being written.

2.3. Hierarchical Proofs

In many cases, a model checker cannot handle a large enough instance of a specification to provide enough confidence in its correctness. When model checking does not suffice, we must use proof. Because TLA⁺ specifications are mathematical formulas, correctness properties are expressed directly as mathematical theorems. The advantage is that there is no need to generate verification conditions; what you see is what you prove.

To prove that a state predicate I is an invariant of a specification S , we must prove $S \Rightarrow \Box I$, where $\Box I$ is the temporal formula asserting that I is always true. When S has the usual form $Init \wedge \Box Next$, this is proved by finding a formula Inv , called an invariant of $Next$, that satisfies $Init \Rightarrow Inv$, $Inv \wedge Next \Rightarrow Inv'$, and $Inv \Rightarrow I$, where Inv' is the formula obtained from Inv by priming all variables.

To prove that a specification S_1 implements another specification S_2 , we must express the variables of S_2 as functions of the variables of S_1 and prove $S_1 \Rightarrow \overline{S_2}$, where $\overline{S_2}$ is formula S_2 with its variables replaced by the corresponding functions of the variables of S_1 . The proof rules of TLA reduce all proofs to reasoning about individual states or pairs of states, using ordinary, nontemporal mathematical reasoning. Over the years, such state-based reasoning has been found to be more reliable than behavioral reasoning, in which one reasons directly about the entire execution.

The theorems that express correctness may be hundreds or thousands of lines long, and their proofs are long and complex. We have developed a hierarchical proof method in which the structure of the formula to be proved largely determines the structure of the proof [2, 6]. For example, the proof of $A \Rightarrow B \wedge C$ consists of the two lower-level steps $A \Rightarrow B$ and $A \Rightarrow C$, and the proof of $A \vee B \Rightarrow C$ consists of the steps $A \Rightarrow C$ and $B \Rightarrow C$.

We have only minimal tool support, in the form of Emacs macros, to help with hand proofs. One interesting application of TLC is to debug these theorems before proving them, and to debug the proofs as they are written. We have not explored the use of mechanical proof systems for real system specifications.

3. Experience

3.1. Wildfire

In the fall of 1996, three of us began a project to verify the cache-coherence protocol of an EV6-based multiprocessor [3]. (EV6 and EV7 are the internal names for the Alpha 21264 and 21364 processors.) We spent about three months writing a TLA⁺ specification of the protocol, starting with a stack of detailed, incomplete, and inconsistent design documents. Interacting frequently with the system's designers by email and telephone, we produced a 1900 line specification. We also wrote a high-level TLA⁺ specification of the Alpha memory model, which the protocol was supposed to implement. The memory model and a simplified version of the protocol specification are available on the web [8].

TLC was not yet written, so our only reasonable option for verifying the protocol was to write a hand proof. We did not have the resources to write a complete proof, so we decided to find

an invariant of the protocol and prove its invariance. We wrote about 1000 lines of informal state predicates that formed the major part of a complete (inductive) invariant. (Although it would have been straightforward, we decided not to spend the time writing the invariant in TLA^+ .) We selected two conjuncts, each about 150 lines long, as the part of the invariant most likely to reveal an error. We completed the proof for one of the conjuncts; it was about 2000 lines long and 13 levels deep. The proof of the second conjunct would have been about twice as long, but we stopped about halfway through because we decided that the likelihood of its discovering an error was too small to justify further effort. We spent about seven months on these two proofs.

We also wrote an informal higher-level proof of one crucial aspect of the protocol. It was about 550 lines long and had a maximum depth of 10 levels.

We found two ways in which the protocol did not implement the Alpha memory model. One was deemed to be an error in the memory model, which was subsequently changed in [1]. The other was a genuine bug in the protocol—an easily-fixed error in one entry of one table. The simplest scenario displaying the bug required four processors, two memory locations, and over 15 messages. We believe that this error could have been found only by writing a proof; an instance of the protocol exhibiting the error would have too many states to be checked exhaustively by a model checker.

Perhaps the major achievement of the project was to subject the protocol to a level of rigorous analysis that significantly increased the designer's confidence in its correctness. The designers were quite happy with our work.

3.2. EV7

Influenced by our effort on the EV6 protocol, engineers decided to use TLA^+ to verify the cache-coherence protocol of the EV7. This project began in the spring of 1998 and is not yet finished. This time, the specification was written by an engineer who received a few hours instruction on TLA^+ . (At the time, there was no language manual.) His specification was about 1800 lines long.

Meanwhile, the TLC model checker was being written. By the fall of 1998, it was ready to be applied to the TLA^+ specification. TLC was able to handle the specification; no modifications were required. The largest instances it was feasible to check with TLC had one cache line, two data values, and three processors. These instances had about 12 million distinct reachable states and originally took several days to run. Improvements to TLC have since reduced that time to a few hours.

We had planned to use TLC to check the RTL implementation by translating runs of the RTL simulator into behaviors at the level of the TLA^+ specification, and using TLC to check those behaviors. That idea was put aside, and is only now being implemented, because the engineers discovered another way to use TLC for RTL-level testing. The error traces that TLC produced in the course of debugging the specification often exhibited corner cases not considered by the designers. So, the verification team translated those traces into input stimuli for the verification team's RTL simulator. The translation was then automated and TLC was used to generate randomly chosen traces. The RTL input from such traces is better

than purely random input because it satisfies the TLA^+ specification.

Work is continuing on combining model checking and simulation more systematically [10]. A translator from output of the RTL simulator to behaviors at the level of the TLA^+ specification is being written. TLC will then be used to check that the RTL code implements the TLA^+ specification during simulations. In the process of checking simulation steps, TLC also collects information about the TLA^+ specification states visited. TLC can then generate traces to interesting but unvisited specification states. These traces can be used to direct the RTL simulation towards coverage gaps.

An invariance proof was also written for the specification. TLC was used extensively to debug the invariant.

As soon as we started using TLC, we found many errors in the TLA^+ specification. Not counting simple mistakes that were easily corrected, we found about 70 errors. About 90% of them were discovered by TLC; the rest were found by a human reading the specification. Most of the errors were introduced when translating from the informal specification; they demonstrate the ambiguity inherent in such specifications. Five design/implementation errors were discovered—one directly by TLC, the other four by using TLC error traces to generate simulator input.

The engineers were quite happy with TLA^+ . The verification group preferred the TLA^+ specification to the informal English one. The EV8 designers were planning to use a TLA^+ specification rather than an English one as the "official" cache-coherence protocol specification. However, the EV8 was cancelled. The engineers from that project, who are now at Intel, are continuing to use TLA^+ . They report that TLA^+ and TLC are starting to become widely used within the former Alpha group at Intel, and they are generating significant interest in other groups.

3.3. Itanium

We have also applied TLA^+ to the cache-coherence protocols for multiprocessors based on the Intel Itanium processor. These systems have components that were designed elsewhere and therefore had to be modeled very abstractly. Simply writing the specifications uncovered many ambiguities in the English descriptions and suggested two small design changes.

TLC could not check the TLA^+ specification of these protocols on large enough instances to give us adequate confidence in the design. This was because (i) the highly abstract models of components designed elsewhere allowed many more interleavings than any actual implementation would, and (ii) the design appeared to require at least four processors to produce interesting scenarios. However, we were able to run meaningful tests by having TLC generate random simulations of large instances of the specification. We are currently exploring ways of model checking larger instances.

As part of this effort, we wrote a TLA^+ specification of the Itanium memory model, which we hope to release in the near future. In addition to serving as a correctness condition for hardware implementations, this specification could be used for analyzing the correctness of software systems intended to run on Itanium-based systems.

In this project, we were separated from the design team by sev-

eral thousand miles. The fact that an engineer could make so much progress with TLA^+ and TLC in relative isolation re-enforced our confidence in the ability of engineers to use these tools.

3.4. Other Projects

We have also applied TLA^+ in a number of smaller projects. Disk Paxos [2] was developed for use in a project; to explain the algorithm in its full generality, we described it to the engineers with a TLA^+ specification. We used TLA^+ and TLC to find bugs in proposals submitted to the working group for the PCI-X bus protocol. For a database system project, we used TLC to check database recovery and cache-management protocols. We now routinely use TLC to check the concurrent algorithms we write in the course of our research.

4. Software

The major industrial applications of TLA^+ have been in the realm of hardware. Hardware engineers routinely use tools based on formal methods to check lower-level designs. They are not accustomed to using tools to check their high-level designs. However, they do write careful informal specifications of those designs. Engineers are aware that, with the increasing complexity of hardware, errors in their high-level designs are becoming an important concern. We have found them receptive to the idea of using TLA^+ at the protocol level.

At the highest level, there is no fundamental difference between hardware and software. The EV6 and EV7 cache-coherence protocols, which send messages that can be reordered in flight, look very much like distributed algorithms in which separate computers communicate over a local area network. So TLA^+ , which has proven useful for hardware, should be just as useful for software. However, there does seem to be a cultural difference between hardware and software engineers. Software engineers do not have the same tradition of relying on specifications that hardware engineers do.

An important lesson we have learned is that moving formal methods from the research community to the engineering community requires patience and perseverance. Engineers are under severe constraints when designing a system. Speed is of the essence, and they never have as many people as they can use. Engineers must be convinced that formal methods will help before they will risk using them.

We plan to explore the use of TLA^+ for specifying and checking the high-level design of concurrent software systems. We believe that TLA^+ is ideal for specifying these systems. We expect that the methods of generating tests and of checking an implementation against the higher-level specification, developed for the EV7, can be used for software as well. We hope to work with engineers to discover how best to use TLA^+ in the software development process.

Acknowledgments

We were assisted by many colleagues in this work. On the EV6 project: Madhumitra Sharma helped us to understand the proto-

col and write its high-level proof; and Paul Harter helped write the low-level proof. On the EV7 project: Joshua Scheid wrote the specification; Homayoon Akhiani and Jonathan Nall implemented the TLA^+ to RTL translator; Damien Doligez wrote the invariance proof; and Serdar Tasiran has been implementing the RTL to TLA^+ translation. On the Itanium project: Jae Yang helped specify the cache-coherence protocol and Gil Neiger helped us specify the Itanium memory model. On other projects: Thomas Rodeheffer worked on the PCI-X bus protocol; and David Lomet worked on the database protocols. Rajeev Joshi helped us to build a satisfiability-based model checker for TLA^+ .

References

- [1] Alpha Architecture Committee. *Alpha Architecture Reference Manual*. Digital Press, Boston, third edition, 1998.
- [2] E. Gafni and L. Lamport. Disk paxos. Technical Report 163, Compaq Systems Research Center, July 2000. To appear in *Distributed Computing*. Currently available on the web at <http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr%20-163.html>.
- [3] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and design of AlphaServer GS320. In A. Gupta, editor, *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 13–24, Nov. 2000.
- [4] L. Lamport. TLA —temporal logic of actions. A web page, a link to which can be found at URL <http://lamport.org>. The page can also be found by searching the Web for the 21-letter string formed by concatenating uid and lamporttlahomepage.
- [5] L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, May 1994.
- [6] L. Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August–September 1995.
- [7] L. Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2002.
- [8] L. Lamport, M. Sharma, M. Tuttle, and Y. Yu. The wildfire verification challenge problem. At URL <http://www.research.compaq.com/SRC/tla/wildfire-challenge.html> on the World Wide Web. It can also be found by searching the Web for the 24-letter string wildfirechallengeproblem.
- [9] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1991.
- [10] S. Tasiran, Y. Yu, B. Batson, and S. Kreider. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *In Proceedings of the 3rd IEEE Workshop on Microprocessor Test and Verification, Common Challenges and Solutions*. IEEE Computer Society, 2002. To appear as an HP technical report.
- [11] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA^+ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Berlin, Heidelberg, New York, Sept. 1999. Springer-Verlag. 10th IFIP wg 10.5 Advanced Research Working Conference, CHARME '99.

Rigour is good for you *and* feasible: reflections on formal treatments of C and UDP sockets

Michael Norrish Peter Sewell Keith Wansbrough

{Firstname.Lastname}@cl.cam.ac.uk

Computer Laboratory, University of Cambridge, Cambridge CB3 0FD

1. Introduction

We summarise two projects that formalised complex real world systems: UDP and its sockets API, and the C programming language. We describe their goals and the techniques used in both. We conclude by discussing how such techniques might be applied to other system software and by describing the benefits this may bring.

2. Specifying UDP and the sockets API

We recently formalised a substantial behavioural specification, that for the Internet protocol UDP, as presented to programmers through the *sockets interface* [12, 10, 1, 5, 11]. Our aim was to make clear the behavioural subtleties of the widely used – but poorly documented – sockets API. This clarification of the interface should ease the production of robust software that uses it. The specification was necessarily developed *post hoc*; we developed it by referring to existing documentation (RFCs and source code, for example), and by experimentally checking existing implementations, using automated tools. We produced the specification in the following three stages:

Syntax We developed a precise notation for describing our systems. We specified abstract syntax with which to denote values corresponding to threads, hosts, messages, sockets, an abstract network, and a simple functional language in which to write programs for threads. Specifying syntax can be seen either as writing a grammar (using BNF notation, say), or as writing data type descriptions in a programming language, particularly one with convenient support for expressing disjoint unions and recursion.

For example, using an ML-like notation, the body of an IP message is specified as in Figure 1. The first possibility of the three above corresponds to a UDP message, with optional destination and source ports, as well as a message body. The remaining two possibilities correspond to the two

types of ICMP message we modelled, unreachable host and port messages respectively, each with source and destination details recorded. This is an abstraction of the structure of actual IP packets: there are many other fields, but they are not significant for programs that use the part of the sockets interface we consider, so need not be modelled.

Typing We developed a set of typing rules, expressing constraints on the values generable from the abstract syntax. For example, our network hosts include a list of active sockets and we require that each of these have a unique file descriptor. Another constraint is that no message in a socket’s incoming queue should include a “martian” IP address. These constraints are both examples of *assertions*, requirements that might be checked dynamically. We also specified a more traditional type system for our miniature programming language.

Behaviour Finally, we described the legal behaviours of the system. Using a labelled transition system (a particular form of automata), we specified the possible ways in which the system might evolve. Our model encompasses threads in hosts, the various non-deterministic behaviours assumed of networks (message dropping, duplication and misordering) and timing requirements.

We illustrate with two examples from the 78 rules that describe the interactions between network hosts, the network and the threads of a process running on a host. Each rule is of the general form

$$h_0 \xrightarrow{\text{label}} h$$

Here h_0 is the state of a network host’s kernel, and *label* describes an interaction between that kernel and either the network or the threads of a user-level process. The result of this interaction is presented in the expression h .

In Figure 2, our first rule describes a thread with identifier *tid* connect-ing a socket to an external address. The evaluation context \mathcal{F} picks out the pertinent components of

```

ipBody = UDP of (port option * port option * string)
| ICMP_HOST_UNRCH of (ip * port option * ip * port option)
| ICMP_PORT_UNRCH of (ip * port option * ip * port option)

```

Figure 1. Type declaration for the body of an IP message

the given host. The *ifds* component is the host's network interfaces. The next two components specify that in the state before the interaction, the host records that thread *tid* is running. After the interaction (thread *tid* making its call), the host records that the host is due to return a unit value (representing success) to the same thread. In the meantime, that thread is blocked waiting for this return to happen.

The main effect of the call is to alter the state of the socket with the file descriptor *fd*, which is the last component of the tuple. It has its destination IP address and destination port (fourth and fifth fields within the SOCK tuple) modified to take on the values of the *i* and *ps* parameters. The source address and port also change, with the source port becoming “autobound” if it is not bound already, and the source address being generated by examining the host's current interfaces (*ifds*) and the destination.

Our second example, in Figure 3, illustrates a host receiving a packet off the network and delivering to a matching socket. The context *S* is an injective function that inserts a socket (here *s*, initially) into an implicit list of other sockets. The rule describes a transition where one of the host's sockets changes, but where the host's other sockets, and its other components all remain unchanged. The label on the transition arrow is the incoming packet. The *lookup* function returns the set of sockets that such a packet could be delivered to, and the other side conditions check that the IP addresses involved are reasonable. The resulting state is updated so that the message queue of socket *s* is extended with the message, paired with interface information about how the message was received.

3. Specifying C

A similar approach was taken in the formalisation of the semantics of C done by the first author [7, 8]. This project was undertaken with the aim of showing that the tools of the theoretical community could be used to model a real world programming language, and do so with high confidence in the correctness of the result.

Syntax The concrete syntax of C expressions and statements is specified in the ISO Standard [6]. Generating an abstract syntax that elided details only necessary for parsing was trivial.

Typing The type system in the standard is carefully described in natural language. The formalised system is non-

trivial only in its slightly complicated treatment of l-values and normal values. For example, the expression *e.fld1*, with the field *fld1* declared as an array, can only be a normal value if the expression *e* is an l-value. There is also a lot of tedious detail in specifying parts of the system like the *usual arithmetic conversions*, which determine the types of the results of binary arithmetic operators.

Behaviour This project did not specify the behaviour of the standard's library. A great deal of complexity remains in the core language, however. In particular, the details of C's sequence points, and the degree to which expression evaluation is allowed to refer to and update objects in memory, are very complicated. These and the standard's three forms of under-determinism, *implementation-defined*, *unspecified* and *undefined* behaviours, were modelled very carefully.

The operational rules used have conclusions of the general form

$$\langle e_0, \sigma_0 \rangle \rightarrow \langle e, \sigma \rangle$$

Here an expression *e*₀, coupled with a state *σ*₀, reduces to a new expression-state pair. States include, among other components, the contents of memory, which parts of memory are allocated, and which parts are initialised.

For example, Figure 4 presents two rules for expressions involving binary operators other than &&, || and , (comma). The first rule can be paraphrased: if *e*₁ can take a step to *e*'₁ (transforming state *σ*₀ to *σ* on the way), then the expression *e*₁ ⊕ *e*₂ can reduce to *e*'₁ ⊕ *e*₂. When the arguments to the operator ⊕ have been fully evaluated, the operator will take effect, reducing *value*₁ ⊕ *value*₂ to the appropriate result.

By way of contrast, there is one rule for &&, allowing only the first argument to be evaluated, which enforces that operator's “short-circuit” behaviour:

$$\frac{\langle e_1, \sigma_0 \rangle \rightarrow \langle e'_1, \sigma \rangle}{\langle e_1 \&\& e_2, \sigma_0 \rangle \rightarrow \langle e'_1 \&\& e_2, \sigma \rangle}$$

C's rules governing the application of side effects are similarly under-determined. Side effects need not be applied as they are generated, nor in order. This under-determinism is tempered by strong constraints, which deem certain sequences of memory references and updates to result in *undefined behaviour*. Programs that might exhibit such sequences are in the same class as those that divide by zero, or access uninitialised memory.

$$\begin{array}{c}
\mathcal{F}(ifds, tid, \text{RUN}_d, \text{SOCK}(fd, *, ps_1, *, *, e, f, mq)) \\
\hline
tid \cdot \text{connect}(fd, i, ps) \\
\hline
\mathcal{F}(ifds, tid, \text{RET}(\text{OK}())_{dsch}, \text{SOCK}(fd, \uparrow i_1, \uparrow p'_1, \uparrow i, ps, e, f, mq))
\end{array}$$

where $\mathbf{F_context}(\mathcal{F}) \wedge i_1 \in \text{outroute}(ifds, i) \wedge p'_1 \in \text{autobind}(ps_1, \mathcal{F})$

Figure 2. Performing a connect call

$$\begin{array}{c}
h \text{ with } sks := \mathcal{S}(s) \\
\hline
\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data)) \\
\hline
h \text{ with } sks := \mathcal{S}(s \text{ with } mq := \text{APPEND } s.mq [(\text{IP}(i_3, i_4, \text{UDP}(ps_3, ps_4, data)), ifd.ifid)])
\end{array}$$

where $\text{socklist_context}(\mathcal{S}) \wedge s \in \text{lookup}(\mathcal{S}(s), (i_3, ps_3, i_4, ps_4)) \wedge$
 $ifd \in h.ifds \wedge i_4 \in ifd.ipset \wedge i_4 \notin \text{LOOPBACK} \wedge$
 $i_3 \notin \text{MARTIAN} \cup \text{LOOPBACK}$

Figure 3. Receiving a packet from the network

$$\frac{\langle e_1, \sigma_0 \rangle \rightarrow \langle e'_1, \sigma \rangle}{\langle e_1 \oplus e_2, \sigma_0 \rangle \rightarrow \langle e'_1 \oplus e_2, \sigma \rangle} \quad \frac{\langle e_2, \sigma_0 \rangle \rightarrow \langle e'_2, \sigma \rangle}{\langle e_1 \oplus e_2, \sigma_0 \rangle \rightarrow \langle e_1 \oplus e'_2, \sigma \rangle}$$

Figure 4. Behaviour of C's binary expressions

4. Reflections

Both projects described above were examples of *post hoc* specification. Our specifications required us to compare our developing models with real world, un-formalised, systems. In the case of C, the un-formalised system was the ISO standard. Being quite carefully written, checking the formal specification manually was not infeasible. In addition, the formalisation was also checked by the proof of a great variety of additional theorems. These were not necessarily particularly deep results, indeed were often quite trivial. When these proofs failed, this pointed out a problem, either in the model, or occasionally, in expectations of how the model should behave.

In the case of UDP sockets, there is no such natural-language standard. The RFCs are not complete, and in any case specify only the wire behaviour not the sockets API behaviour. In addition to consulting documentation, we used experimental techniques to determine the behaviour of implementations in particular situations. These experiments were important in determining behaviours in extreme cases that were often not very well documented.

In both cases, we also used the HOL interactive theorem-proving system [3, 9]. For the C semantics, HOL was used from the outset, enabling the definition of a type representing C syntax, and the appropriate relations, for typing and behaviour, over those types. Used entirely in this “definitional capacity”, HOL provides an expressive, statically type-checked language in which to write logical definitions. HOL’s theorem-proving facilities were subsequently used to prove the theorems referred to above.

With UDP, we found HOL’s greatest benefit to be the sanity checks we were able to make of the specification. The mechanisation of what was originally a pen-and-paper specification caught numerous errors. Most of these were typographical errors, such as using functions or predicates with inappropriate types or numbers of arguments. Nonetheless, we also later proved that our behavioural system preserved an important invariant (or safety) property. Attempting this proof caught a number of omissions and errors in the original model, drawing our attention to its obscure corners.

We have yet to marry our specifications of UDP and C to verification, in any significant way. Nonetheless, our specifications are valuable in themselves. We have given formal description to large and complicated systems, and in so doing, have made explicit the contract between users and implementors of those systems. Our (annotated!) specifications provide detailed documentation which may be useful to both users and implementors of UDP sockets and C compilers. For UDP, and with mechanical assistance, we have explored both the state spaces of the formal system, and also of the deployed implementations, discovering, for example,

how the Windows and Linux implementations differ.

This is a general phenomenon: while underway, the specification activity brings benefits of increased understanding. Later, the specification is useful as precise documentation. Subsequent program verification, while desirable, remains infeasible in most contexts.

5. Lessons—what others can do

Language designers give rigorous specifications of language syntax as a matter of course. Moreover, there are long-established formal notations for the specification of concrete syntax, and tools to support the use of these notations. Syntax is specified not just for programming languages, but other formal languages such as HTML and XML.

Rigorous definitions of type systems is less common, though standard practice in the programming language and semantics communities. These definitions are usually expressed in informal mathematics, using a body of widely-accepted stylised idioms, rather than in any particular logical system. General tool support is lacking, though interactive proof assistants such as HOL are increasingly used in large-scale work. In addition to the often elaborate type systems constructed around research calculi and languages, types have been usefully deployed in, for example, the Xduce project [4], which defines regular expression types for manipulating XML data, and the Hafnium project [2] which used type inference to address the Y2K problem for COBOL programs.

The situation for behavioural specification is even worse. A very weak form is the use of assertions, becoming more widespread. “State-of-the-art” in most fields is more-or-less precise natural language. Network protocols at lower levels of abstraction are specified precisely, in RFCs and other documentation, and some hardware components do have formal specifications, but we are unaware of significant examples at the systems level.

We do not propose a universal methodology for writing formal specifications to address these problems. Indeed, we believe that no such methodology is pragmatically viable. Our experience suggests, however, that a number of important principles apply:

Specify in small pieces. Do not set out to specify everything all at once, nor in exhaustive detail. Attack tractable pieces of systems, at appropriate levels of detail, and begin with simple aspects such as typing requirements.

Choose the right pieces. Formal specification is most valuable for systems with subtle behaviour, for which it is hard to develop a sufficiently good informal understanding, and (obviously) for systems for which cor-

rectness matters. Concurrent systems, including communication and security protocols, often benefit.

Use intellectual tools. Draw on techniques from appropriate fields, such as operational semantics, type theory, programming language theory, and concurrency theory. Take an existing specification language/idiom or build a new one if required, as appropriate to the problem. Make sure, however, that whatever specification notation is used has a precise meaning.

Use mechanical tools. Use a notation with tool support, so that mechanical analysis and checking of the specification is possible. This might be simple type-checking, model-checking, machine-assisted theorem proving, or automated testing.

Begin as early as possible. *Post hoc* specification activities such as the two we described here have value, but if used at the design stage then rigorous treatments can be valuable in other ways, providing early understanding and a push towards clean design.

In the long term, we believe greater rigour is essential in the development of more robust software at all levels; there are many behaviourally subtle aspects of operating systems which could benefit from it, and for which the tools are now available.

References

- [1] U. CSRG. 4.2BSD, 1983.
- [2] P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sørensen, and M. Tofte. AnnoDomini: from type theory to year 2000 conversion tool. In *POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.
- [3] M. J. C. Gordon and T. Melham, editors. *Introduction to HOL: a theorem proving environment*. Cambridge University Press, 1993.
- [4] H. Hasoya and B. C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80, 2001.
- [5] IEEE. *Information Technology—Portable Operating System Interface (POSIX)—Part xx: Protocol Independent Interfaces (PII)*, P1003.1g. Mar. 2000.
- [6] *Programming languages – C*, 1990. ISO/IEC 9899:1990.
- [7] M. Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.
- [8] M. Norrish. Deterministic expressions in C. In S. D. Swierstra, editor, *Programming languages and systems, 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 147–161. Springer, March 1999.
- [9] M. Norrish and K. Slind. A thread of HOL development. *Computer Journal*, 45(1):37–45, 2002.

- [10] A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proceedings of TACS 2001: Theoretical Aspects of Computer Software (Sendai, Japan)*, LNCS 2215, pages 535–559, Oct. 2001.
- [11] W. R. Stevens. *UNIX Network Programming Vol. 1: Networking APIs: Sockets and XTI*. Prentice Hall, second edition, 1998.
- [12] K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing UDP: Mechanized semantics for sockets, threads and failures. In *Programming languages and systems, 11th European Symposium on Programming*, Lecture Notes in Computer Science. Springer, 2002. To appear.

Capturing OS Expertise in an Event Type System: the Bossa Experience

Julia L. Lawall

DIKU, University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

Gilles Muller

Ecole des Mines de Nantes
44307 Nantes Cedex 3, France
Gilles.Muller@inria.fr

Luciano Porto Barreto

COMPOSE group, INRIA/LaBRI
33402 Talence Cedex, France
barreto@labri.fr

Abstract

Emerging applications have increasingly specialized scheduling requirements. Changing the scheduling policy of an existing OS is, however, often difficult because scheduling code is typically deeply intertwined with the rest of the kernel. We have recently introduced the Bossa framework to facilitate the implementation and integration of new scheduling policies. While the use of Bossa simplifies the problem of implementing a new scheduler, knowledge of the control and data flow through the scheduling actions of the kernel is still needed to ensure that the behavior of the provided scheduling policy matches kernel expectations. In this paper, we propose a modular type system that provides a high-level characterization of the aspects of kernel behavior that affect the correctness of a scheduling policy. These types guide policy development and are linked with the compiler to enable static verification of correctness properties.

1. Introduction

The need for application-specific scheduling strategies is now well recognized in the OS community, as is demonstrated by the number of recent studies in this area [1, 6, 11, 14, 16, 18, 20]. Nevertheless, developing a new scheduling policy and integrating it into an existing OS is complex, because it requires understanding the (often implicit) conventions used by the OS implementation. One technique that can address this issue is the use of a framework to structure the implementation of a scheduler and make precise its interface with the kernel. Another technique is the use of a *domain-specific language* (DSL) for the development of new policies. A DSL is a programming language providing high-level abstractions appropriate to a given domain. Expertise captured in the language allows policies to be expressed in an intuitive and high-level manner, permits verification, and allows generation of efficient code that is

automatically integrated in the target system. DSLs have demonstrated their interest in a variety of OS subsystems such as network protocols [19], memory coherency protocols [7] and drivers [13].

We have recently introduced the Bossa event-based framework for implementing schedulers [3]. The framework includes a DSL for writing scheduling policies. In the Bossa framework, each scheduling-related action in the kernel, referred to as a *scheduling point*, is replaced by the notification of a Bossa event. A scheduling policy is implemented as a collection of event handlers that are written in the Bossa DSL and translated to a C file by a dedicated compiler. The use of the high-level scheduling abstractions built into the Bossa DSL permits scheduling-specific verifications and optimizations.

Correctness of the Bossa implementation of a scheduling policy requires that the definition of each event handler be consistent with the expectations of the corresponding kernel scheduling points. Because kernel properties vary from OS to OS, knowledge of these expectations cannot be simply built into the Bossa compiler. In this paper, we propose to describe OS behavior using a collection of *event types* that describe the possible process states at each scheduling point and the corresponding expected effect of each event notification. Event types are written by an expert in the target OS. They are provided to the policy developer to guide policy design and implementation, and linked with the Bossa compiler to produce a compiler specialized to a particular OS. This specialized compiler uses knowledge of the DSL abstractions and the event types to check that the handlers of the implementation of a scheduling policy are consistent with kernel expectations.

The rest of this paper is organized as follows. In Section 2, we present the Bossa framework and DSL. Section 3 presents the use of event types to model OS behavior. Section 4 assesses the impact of event types on policy development and optimization. Finally, Section 5 considers related work and Section 6 concludes. We use the Linux 2.2 kernel and its scheduling policy as examples throughout the paper.

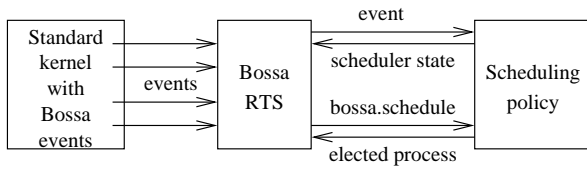


Figure 1. Bossa architecture

2. Bossa

We first present the Bossa framework as a whole, and then focus on the Bossa DSL, which is used to implement new scheduling policies.

2.1. The Bossa framework

The goal of the Bossa framework is to separate the implementation of the scheduler from the rest of the kernel, so that the scheduling policy can easily be changed at the time of kernel recompilation. The Bossa framework replaces scheduling actions in the kernel, such as the modifying of a process state or the electing of a new process, by Bossa event notifications. Events notifications are processed by the Bossa run-time system (RTS), which invokes the appropriate handler defined by the scheduling policy. Bossa events are organized into a hierarchy, according to the function and source of the event. A policy can provide a single event handler for all events related to a particular function, such as a `block.*` handler that applies to all blocking events, or handlers for specific events, such as blocking when waiting for a file or memory page. The selected handler returns the state of the scheduler, indicating whether there is a running process, or there are ready processes available. This information is used at the next `bossa.schedule` event notification: The RTS only invokes the `bossa.schedule` handler of the policy if the current scheduler state indicates that there is no running process and a ready process is available. Otherwise if there is a running process, the RTS allows it to continue, and if there are no running or ready processes, the RTS runs the kernel idle loop.

The RTS is fixed and provided by the framework. The scheduling policy is provided by the policy developer, and implemented using the Bossa DSL. Both the RTS and the compiled scheduling policy are integrated into the kernel. Our performance results, reported in detail elsewhere [4], typically show no penalty for the use of Bossa, on both standard OS benchmarks and realistic applications such as compilation of the Linux kernel.

2.2. The Bossa DSL

A Bossa scheduling policy declares: (i) a collection of scheduling-related structures to be used by the policy, (ii) a set of event handlers, and (iii) a set of interface functions, allowing users to interact with the scheduler. Figure 2 shows some of the declarations made by the Bossa implementation of the Linux 2.2 policy. The `process` declaration lists the policy-specific attributes associated with each process. As reflected by the `policy` field, the Linux 2.2 scheduling policy manages FIFO and round-robin real-time processes, as well as non-real-time processes. The other fields of the process structure are used to determine the current priority of the associated process. The `states` declaration lists the sets of states that are manipulated by the scheduling policy: `running`, `ready`, `yield`, `blocked`, and `terminated`. Each state is described by a class, which indicates the schedulability of processes in the state. The state in the `RUNNING` class contains the currently running process. The states in the `READY` class contain processes that are eligible to be elected at the next context switch. The states in the `BLOCKED` class contain processes that are not eligible to be elected at the next context switch. The states in the `TERMINATED` class represent terminating, and thus no longer schedulable, processes. Each state is also associated with a data structure: either a process variable (`process`) or a queue (`queue`).¹ One `READY` state is designated as sorted; only processes in this state can be elected. Finally, the `ordering_criteria` declaration specifies how the relative priority of processes is computed.

Figure 3 shows the definitions of several event handlers of the Linux 2.2 policy: `system.clocktick`, `block.*`, and `unblock.*`. The effect of the `system.clocktick` handler depends on whether the target process is a real-time process, and if so, on the process's real-time policy. A clock tick has no effect on a FIFO real-time process. If a non-FIFO process has used up its ticks, it is moved to either the `ready` queue or the `expired` queue. Otherwise, the handler decrements the number of ticks remaining. The `block.*` handler moves the target process to the `blocked` queue. The `unblock.*` handler moves the target process from the `blocked` queue to the `ready` queue, if the target process is indeed blocked.

2.3. Assessment

The use of the Bossa framework isolates all policy-specific scheduling actions into a single file implementing a well-defined interface, thus simplifying understanding of the policy as a whole. The use of the Bossa DSL further permits the implementation of the policy to be expressed in

¹No data structure is associated with `terminated`, as terminating processes need not be accounted for by the scheduler.

```

type policy_t =
  enum {SCHED_FIFO, SCHED_RR, SCHED_OTHER};

process = {
  policy_t policy;
  int      rt_priority;
  time     priority;
  time     ticks;
  system struct ctx mm;
}

states = {
  RUNNING running : process,
                previous old_running;
  READY ready      : sorted queue;
  READY expired    : queue;
  BLOCKED yield    : process;
  BLOCKED blocked  : queue;
  TERMINATED terminated;
}

ordering_criteria = {
  highest rt_priority, highest ticks,
  highest ((mm == old_running.mm) ? 1:0)
}

```

Figure 2. Declarations of the Linux 2.2 scheduling policy

a concise and high-level manner. The approach facilitates policy evolution, because the Bossa DSL compiler ensures the consistency of the various parts of the scheduling policy. Nevertheless, the framework and the DSL alone are not sufficient to ensure that the behavior of a given scheduling policy matches kernel expectations.

3. Modeling OS behavior

To ensure that the event handlers of a scheduling policy are consistent with the kernel behavior, the DSL compiler must be aware of the control and data flow through the scheduling points in the kernel. To provide this information, our description of OS behavior consists of two parts: a collection of *event sequences* that describe possible sequences of Bossa events, and a collection of *event types* that describe possible inputs and corresponding outputs for each scheduling event handler. These are interdependent, in that the provided event types must describe a behavior for all inputs that can occur along control paths allowed by the event sequences. We first present event sequences and event types, and then show how this information is used in the Bossa DSL compiler.

```

On system.clocktick {
  if (running.policy != SCHED_FIFO) {
    if (running.ticks == 0) {
      running.ticks = running.priority;
      if (running.policy == SCHED_RR) {
        running => ready;
      }
    }
    else {
      running => expired;
    }
  }
  else {
    running.ticks--;
  }
}

On block.* {
  e.target => blocked;
}

On unblock.* {
  if (e.target in blocked) {
    e.target => ready;
  }
}

```

Figure 3. Selected event handlers of the Linux 2.2 policy

3.1. Event sequences

Linux 2.2 kernel code exhibits a form of pseudo-parallelism, in that execution of a system call by an application can be interrupted at any point by execution of interrupt handlers. We use automata to describe the sequences of events explicitly notified during the treatment of a system call and during the treatment of an interrupt, and then consider the possible interactions between the sequences described by these automata.

In the Linux 2.2 kernel, several common patterns of scheduling events occur in the implementation of system calls. These patterns are described by the automaton shown in Figure 4 (double-circles in the automaton represent the point at which control returns from the system call to the application). The top two branches describe the behavior of system calls that perform at most one scheduling event, such as a request to yield the processor, possibly followed by a `bossa.schedule` event. The remaining branches describe event sequences used by system calls that repetitively yield the processor while waiting for access to a resource.

As in many Unix systems, the treatment of an interrupt in the Linux 2.2 kernel is divided into a very short interrupt

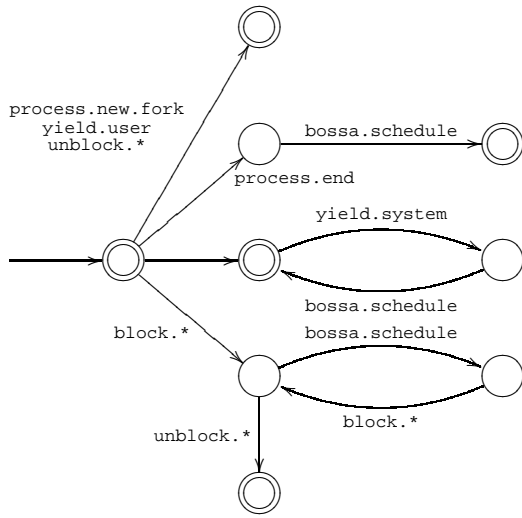


Figure 4. Automaton of Bossa events occurring in system calls

handler, which runs with interrupts disabled, followed by more complex *bottom-half* code, which runs with interrupts enabled. Scheduling points only occur in bottom-half code. The automaton shown in Figure 5 describes the possible sequences of Bossa events used at these scheduling points:

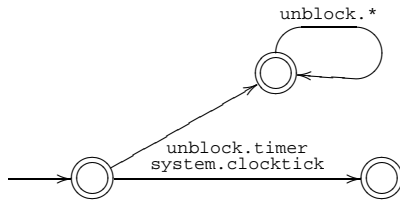


Figure 5. Automaton of Bossa events occurring in interrupts

The complete set of event sequences consists of any sequence allowed by any interleaving of any number of instances of the automata. For example, given the two automata above, `system.clocktick` \rightarrow `block.*` \rightarrow `unblock.*` \rightarrow `bossa.schedule` \rightarrow `block.*` \rightarrow `unblock.*` is a valid event sequence.

3.2. Event types

Event sequences describe the possible control flow through the event handlers of a scheduling policy. Complementarily, event types describe the possible inputs of each

event handler, and the expected result for each such possible input. Event types are classified as those that describe OS behavior local to a single system call or a single interrupt handler, and those that describe the expected event handler behavior taking into account the effects of event sequences on process states.

Event types describing local behavior. An event type specifies the following properties: (i) the emptiness or non-emptiness of the states associated with each class, (ii) the classes associated with the states of the source and target processes of the event, and (iii) the permitted and required movement of processes between classes during the handler. An event can be associated with multiple types, reflecting the various possible states of processes on entry to the event handler and the various effects that the event handler is allowed to produce. For example, `system.clocktick` should intuitively either leave the running process in its current state, if the process has ticks remaining, or preempt the process if the process has exhausted its time slice; `block.*` should move the running process to a BLOCKED state; `unblock.*` should move the target process from a BLOCKED state to a READY state. These behaviors are described by the following event types:

```
system.clocktick:
  <target  $\in$  RUNNING>  $\rightarrow$  <target  $\in$  RUNNING>
  <target  $\in$  RUNNING>  $\rightarrow$  <target  $\in$  READY>
  <target  $\in$  RUNNING>  $\rightarrow$  <target  $\in$  BLOCKED>
```

```
block.*:
  <target  $\in$  RUNNING>  $\rightarrow$  <target  $\in$  BLOCKED>
```

```
unblock.*:
  <target  $\in$  BLOCKED>  $\rightarrow$  <target  $\in$  READY>
```

In addition to describing OS properties, event types can also be used to express constraints of the Bossa framework. In particular, the Bossa framework ensures that the `system.clocktick` handler is only called when there is a running process, and the `bossa.schedule` handler is only called when there is no running process. While these constraints could be built into the Bossa compiler, expressing them at the level of event types makes all constraints on handler behavior accessible to the policy developer within a uniform framework.

Event types describing event sequence behavior. The Bossa state of a process reflects the schedulability of a process, rather than whether or not the process is currently running. For example, a `block.*` event places the target process in a BLOCKED state, indicating that the process is not currently schedulable, but it is not until the next

`bossa.schedule` event that the process is actually preempted. Because state changes can occur in both system calls and interrupts, and the actions of system calls and interrupts are unknown to each other, the Bossa state of a process can be different than the state expected by the context in which the process is used. Event types must thus take into account all possible states of the source and target processes that can result from event sequences.

We first consider how a process can have an unexpected Bossa state in the treatment of a system call. In the Linux 2.2 kernel it is not possible to switch to a new process during bottom half code. Consequently, an interrupt bottom half that changes the state of the running process to indicate preemption does not actually affect the running process until the next `bossa.schedule` event, which triggers a context switch. Other events that can be performed by the application before this `bossa.schedule` event must thus take into account the possibility that the currently running process is not actually in the `RUNNING` state. The Linux 2.2 kernel avoids this issue because the operation of updating a process state is independent of the current state of the process. This approach, however, is difficult to understand and error-prone, because it relies on implicit conventions about the possible state of the affected process at the point of the state change. Bossa event types characterize both the source and target state of each allowed transition. This approach improves readability and safety of a policy, but implies that multiple, sometimes unintuitive, cases must be explicitly accounted for.

As an example of the interaction between interrupts and system calls, consider an event sequence beginning with a `system.clocktick` event in an interrupt bottom half followed by a `block.*` event in a system call. The type of `system.clocktick` allows the handler to move the process in the `RUNNING` state to a `RUNNING`, `READY`, or `BLOCKED` state. The event type of `block.*` must thus describe the expected behavior when the target of the event is in a state associated with any of these classes. The complete set of event types for `block.*` is thus:

$$\begin{aligned} &\langle \text{target} \in \text{RUNNING} \rangle \rightarrow \langle \text{target} \in \text{BLOCKED} \rangle \\ &\langle [] = \text{RUNNING}, \text{target} \in \text{READY} \rangle \\ &\quad \rightarrow \langle \text{target} \in \text{BLOCKED} \rangle \\ &\langle [] = \text{RUNNING}, \text{target} \in \text{BLOCKED} \rangle \\ &\quad \rightarrow \langle \text{target} \in \text{BLOCKED} \rangle \end{aligned}$$

The inconsistency between the expected state of a process and its Bossa state can also occur in the treatment of an interrupt. To obtain access to a resource, one strategy used in the Linux 2.2 kernel is for a process first to add itself to a wait queue, then to test the availability of the resource, and finally to block if the resource is not available. In this case, the interrupt that unblocks waiting processes when the resource becomes available may actually find that a wait-

ing process has not yet blocked, and thus is still running.² Thus, the event type for `unblock.*` must account for the case where the target process is in the `RUNNING` state. Furthermore, because `system.clocktick` followed by `unblock.*` is a valid event sequence, the event type for `unblock.*` must account for the case where the target process is placed by `system.clocktick` in a `READY` or `BLOCKED` state as well. The complete set of event types for `unblock.*` is thus:

$$\begin{aligned} &\langle \text{target} \in \text{BLOCKED} \rangle \rightarrow \langle \text{target} \in \text{READY} \rangle \\ &\langle \text{target} \in \text{RUNNING} \rangle \rightarrow \langle \text{target} \in \text{RUNNING} \rangle \\ &\langle \text{target} \in \text{READY} \rangle \rightarrow \langle \text{target} \in \text{READY} \rangle \\ &\langle \text{target} \in \text{BLOCKED} \rangle \rightarrow \langle \text{target} \in \text{BLOCKED} \rangle \end{aligned}$$

The set of event types induced by the event sequences for `unblock.*` illustrates a case where the event sequences require adding a type for an input configuration for which a type based on local behavior is already available. The `system.clocktick` event can place the target process in a `BLOCKED` state, but this `BLOCKED` state is not necessarily one that represents processes waiting for a resource, and thus to be moved to a `READY` state by `unblock.*`. The new type $\langle \text{target} \in \text{BLOCKED} \rangle \rightarrow \langle \text{target} \in \text{BLOCKED} \rangle$ allows the `unblock.*` handler to leave such processes in their current state. This example illustrates the trade-off between the goal of a precise description of OS behavior and the goal of a description that is applicable to diverse scheduling policies.

The complete set of event types for the Linux 2.2 kernel is shown in Appendix A.

3.3. Event sequences and event types in the Bossa compiler

Bossa provides an event-type compiler to be used by the OS expert and a policy compiler to be used by the policy designer, as illustrated by Figure 6. The event-type compiler checks that the event sequences and the event types are mutually consistent. The policy compiler uses the event types and event sequences to verify and optimize the policy according to the kernel behavior and to produce the corresponding C code.

The event-type compiler first checks that each of the automata is consistent with the event types describing the corresponding local behavior (*i.e.*, in each sequence described by an automaton, the output of one event must be an acceptable input for the next event). The compiler then checks the completeness of the provided types. For this purpose it propagates the input configuration of each provided type

²This case is also checked for by the Linux 2.2 kernel, which treats an `unblock` event differently depending on whether the target is blocked or already in the runqueue.

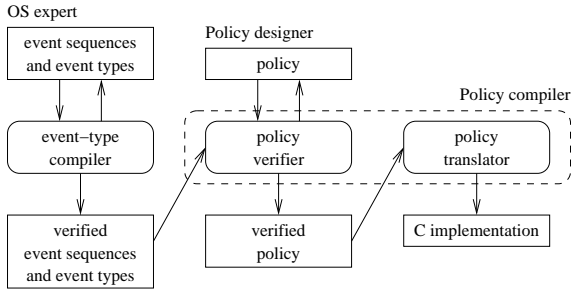


Figure 6. Structure of the Bossa DSL compiler

through the intervening possible event sequences to identify all possible states of the source and target processes at the point at which the event is actually called, and checks that an event type is provided for each possible case. If this step fails, the event-type compiler returns to the OS expert a list of the input configurations for which no event type is provided, as well as a possible event sequence leading to each such configuration. When verification of the event sequences and event types succeeds, the event type compiler produces a representation of this information suitable for linking with the policy compiler.

Once the event sequences and event types are accepted by the event-type compiler, they are used to create an instance of the compiler for the targeted OS. For a given scheduling policy, the Bossa compiler performs an interprocedural analysis of the specified event handlers, guided by the event sequences, and checks that each event handler implements the behavior required by its event types on all reachable inputs. The policy compiler then translates the policy into C code. Information about process states derived from the event types is used to eliminate unnecessary tests in the generated code.

4. Verification and optimization

The main benefit of event types is to enable error detection. In our experience, the interactions captured by the event sequences are easily overlooked, and the verifications performed by the Bossa compiler based on event types detected several missing cases in our initial implementation of the Linux 2.2 policy. The event types also enable the compiler to eliminate some unnecessary tests. In the Linux 2.2 policy, these primarily affect the calculation of the scheduler state. As shown in Appendix A, in several of the event types derived from the event sequences the `RUNNING` class is known to be empty. In these cases, the compiler can statically determine that the scheduler state cannot be `RUNNING`. The use of the event sequences by

the compiler allows the compiler to avoid analysis with respect to input configurations that are allowed by the event types but are not reachable in a given policy. For example, the `yield.user` handler of the Linux 2.2 policy, shown below, can place a process in the `yield` process variable. This operation is only valid if `yield` is empty, but the event type of `yield.user` does not give any information about `BLOCKED`, which is the class of the `yield` variable. The event sequences enable the compiler to determine that `yield` is always empty when the code `running => yield` is executed.

```

On yield.user {
  if (!empty(READY) &&
      e.target in running) {
    if (running.policy == SCHED_OTHER) {
      running => yield;
    }
    else {
      running => ready;
    }
  }
}

```

5. Related Work

Recently, there has been growing interest in new approaches to verification of operating systems code, including the introduction and checking of assertions [10], verification that common sequences of operations are used in an expected way [9], as well as more complex techniques such as model checking [5, 15] and theorem proving [12]. Nevertheless, all of these approaches are limited by the lack of domain-specific knowledge, and thus exhaustive static verification of important properties is often infeasible.

The work on Composable Execution Environments (CEE) includes a logic for specifying scheduling-specific properties [17]. The goal of this logic is to check whether undesirable interactions, such as deadlocks, can occur when a given combination of existing schedulers manage a particular set of tasks. Our goal is orthogonal: to describe the interface between an existing Bossa-ready scheduler and new scheduling policies.

The integration of a Bossa policy into an OS is analogous to the weaving performed by an aspect language [2]. The techniques for describing the behavior of the target system proposed here could be useful to verify the correctness of systems constructed using aspects, in particular aspect-oriented OS components [8].

6. Conclusion

In this paper, we have presented a modular type system describing OS properties for use with the Bossa DSL. The

information contained in these types significantly increases the amount of verification that can be performed at compile time, thus providing confidence in the correctness of the policy. Some compiler optimizations are also enabled. The separation of these types from the language implementation should facilitate the porting of Bossa to other operating systems. A port to Linux 2.4 is currently underway.

References

- [1] A. Atlas and A. Bestavros. Design and implementation of statistical rate monotonic scheduling in KURT Linux. In *IEEE Real-Time Systems Symposium*, pages 272–276, 1999.
- [2] L. P. Barreto, R. Douence, G. Muller, and M. Südholt. Programming OS schedulers with domain-specific languages and aspects: New approaches for OS kernel engineering. Int. Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD, April 2002.
- [3] L.P. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *10th International Conference on Real-Time Systems (RTS'2002)*, pages 19–31, Paris, France, March 2002.
- [4] L.P. Barreto, G. Muller, J.L. Lawall, and K. Kono. A framework for simplifying the development of kernel schedulers: design and performance evaluation. Technical Report 02/8/INFO, Ecole des Mines de Nantes, 2002.
- [5] A. Basu, M. Hayden, G. Morrisett, and T. von Eicken. A language-based approach to protocol construction. In *Proceedings of the ACM SIGPLAN Workshop on Domain Specific Languages*, Paris, France, January 1997.
- [6] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportional fair scheduling in multiprocessor servers arbitrary deadlines. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'2001)*, Taipei, Taiwan, May 2001.
- [7] S. Chandra, B. Richards, and J.R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 237–248, 1996.
- [8] Y. Coady, G. Kiczales, J.S. Ong, A. Warfield, and M. Feeley. Brittle systems will break – not bend: Can aspect-oriented programming help? In *Proceedings of the Tenth ACM SIGOPS European Workshop (EW'2002)*, Saint-Emilion, France, September 2002. To appear.
- [9] A. Engler, D. Yu, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72, Banff, Canada, October 2001.
- [10] T. Jim, Morrisett G, Grossman D, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.
- [11] I. Kouvelas and V. Hardman. Overcoming workstation scheduling problems in a real-time audio tool. In *Proceedings of the USENIX 1997 Conference*, pages 235–242, Anaheim, CA, January 1997.
- [12] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.
- [13] F. Mérellon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 17–30, San Diego, California, October 2000.
- [14] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 184–197, October 1997.
- [15] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS scheduler kernel. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 488–497, Limerick, Ireland, June 2000.
- [16] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *18th ACM Symposium on Operating Systems Principles*, pages 89–102, October 2001.
- [17] J. Regehr, A. Reid, K. Webb, and J. Lepreau. Composable execution environments. Flux group technical note 2002-02, University of Utah, May 2002.
- [18] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI-99)*, pages 145–158, February 1999.
- [19] S. Thibault, J. Marant, and G. Muller. Adapting distributed applications using extensible networks. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 234–243, Austin, Texas, May 1999. IEEE Computer Society Press.
- [20] D. K. Y. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, 5(4):475–488, August 1997.

A. Event types for the Linux 2.2 kernel

Event types are classified as being derived from local behavior or being derived from interactions due to event sequences. The type of a hierarchical event applies to all more specific events for which no other information in that section is provided. For example, the types based on event sequences for `yield.*` also apply to the `yield.user`, `yield.system.pause`, and `yield.system.immediate` events.

A.1 Event types describing local behavior

```

process.new.* :
  [tgt in NOWHERE] -> {[tgt in READY], [tgt in BLOCKED]}

process.new.fork :
  [src in RUNNING, tgt in NOWHERE] -> {[src in RUNNING, tgt in READY],
    [src in RUNNING, tgt in BLOCKED]}

process.new.initial_process :
  [[] = RUNNING, tgt in NOWHERE] -> {[[] = RUNNING, tgt in READY],
    [[] = RUNNING, tgt in BLOCKED]}

process.end : {[tgt in BLOCKED], [tgt in READY]} -> [tgt in TERMINATED]

process.end : [tgt in RUNNING] -> [tgt in TERMINATED]

yield.* :
  [tgt in RUNNING] -> { [tgt in READY], [tgt in BLOCKED], [tgt in RUNNING] }

yield.system.* :
  [tgt in RUNNING] -> { [tgt in READY], [tgt in BLOCKED], [tgt in RUNNING] }

yield.system.pause :
  [tgt in RUNNING] -> { [tgt in READY], [tgt in BLOCKED], [tgt in RUNNING] }

yield.system.immediate :
  [tgt in RUNNING] -> { [tgt in READY], [tgt in RUNNING] }

yield.user :
  [tgt in RUNNING] -> { [tgt in READY], [tgt in BLOCKED], [tgt in RUNNING] }

system.clocktick :
  [tgt in RUNNING] -> { [tgt in READY], [tgt in BLOCKED], [tgt in RUNNING] }

block.* : [tgt in RUNNING] -> [tgt in BLOCKED]

unblock.* : [tgt in RUNNING] -> [tgt in RUNNING]

unblock.* : [p in RUNNING, tgt in BLOCKED]
  -> { [[p,tgt] in READY],
    [p in RUNNING, tgt in READY],
    [p in RUNNING, tgt in BLOCKED] }

unblock.* : [[] = RUNNING, tgt in BLOCKED]
  -> { [[] = RUNNING, tgt in READY],
    [[] = RUNNING, tgt in BLOCKED] }

unblock.synchronous_wake_up_process : [tgt in RUNNING] -> [tgt in RUNNING]

unblock.synchronous_wake_up_process :
  [tgt in BLOCKED] -> { [tgt in READY], [tgt in BLOCKED] }

unblock.wake_up_process : [tgt in RUNNING] -> [tgt in RUNNING]

unblock.wake_up_process : [p in RUNNING, tgt in BLOCKED]
  -> { [[p,tgt] in READY],
    [p in RUNNING, tgt in READY],
    [p in RUNNING, tgt in BLOCKED] }

```

```

unblock.wake_up_process : [[] = RUNNING, tgt in BLOCKED]
  -> { [[] = RUNNING, tgt in READY],
    [[] = RUNNING, tgt in BLOCKED] }

unblock.timer : [tgt in RUNNING] -> [tgt in READY]

unblock.timer : [tgt in READY] -> [tgt in READY]

unblock.timer : [p in RUNNING, tgt in BLOCKED] ->
  { [p in RUNNING, tgt in BLOCKED],
    [p in RUNNING, tgt in READY],
    [[p,tgt] in READY] }

unblock.timer : [[] = RUNNING, tgt in BLOCKED]
  -> { [[] = RUNNING, [tgt] in READY],
    [[] = RUNNING, [tgt] in BLOCKED] }

bossa.schedule : [[] = RUNNING, q in READY] -> [q in RUNNING, READY!]
}

```

A.2 Event types describing event sequence behavior

```

unblock.timer :
  [tgt in TERMINATED] -> [tgt in TERMINATED]

unblock.* :
  [tgt in TERMINATED] -> [tgt in TERMINATED]

unblock.* :
  [tgt in READY] -> [tgt in READY]

yield.* :
  [[] = RUNNING, tgt in BLOCKED] -> [tgt in BLOCKED]

yield.* :
  [[] = RUNNING, tgt in READY] -> { [tgt in READY], [tgt in BLOCKED] }

unblock.wake_up_process :
  [tgt in READY] -> [tgt in READY]

unblock.synchronous_wake_up_process :
  [tgt in READY] -> [tgt in READY]

process.new.fork :
  [tgt in NOWHERE, [] = RUNNING, src in BLOCKED] ->
    {[src in BLOCKED, tgt in READY], [[src,tgt] in BLOCKED]}

process.new.fork :
  [tgt in NOWHERE, [] = RUNNING, src in READY] ->
    {[[src,tgt] in READY], [src in READY, tgt in BLOCKED]}

block.* :
  [[] = RUNNING, tgt in BLOCKED] -> [tgt in BLOCKED]

block.* :
  [[] = RUNNING, tgt in READY] -> [tgt in BLOCKED]

```

Session 4. Robust Service

1. Overload Management as a Fundamental Service Design Primitive

Authors: Matt Welsh, David Culler

page 62

2. Rewind, Repair, Replay: Three R's to Dependability

Authors: Aaron B. Brown, David A. Patterson

page 70

Overload Management as a Fundamental Service Design Primitive

Matt Welsh and David Culler

Computer Science Division

University of California, Berkeley

{mdw,culler}@cs.berkeley.edu

Abstract

This position paper makes the case that overload management should be a critical design goal for Internet-based systems and services. Few Internet service designs take overload into account, treating the problem as one of capacity planning rather than engineering the service to behave gracefully under extreme load. We argue that the right approach to overload management is to explicitly signal overload conditions to the application, allowing it to participate in resource management decisions. Furthermore, we claim that feedback-driven control, rather than static resource limits, should be the basis for detecting and controlling overload. We present a feedback-driven approach to overload control based on the staged event-driven architecture (SEDA) model for Internet service design. This approach makes use of adaptive admission controllers for meeting administrator-specified performance targets, such as 90th percentile response time. We demonstrate the use of these overload control mechanisms in two applications: a complex Web-based e-mail service, and a dynamic Web server benchmark.

1 Introduction

In this position paper we argue that overload prevention is a fundamental requirement for distributed systems and services connected to the Internet. Unfortunately, few systems have adequately addressed the management of extreme load, relying mainly on overprovisioning of resources (e.g., replication). However, given the enormous user population on the Internet, overprovisioning is infeasible as the peak load that a service experiences may be orders of magnitude greater than the average. The events of September 11, 2001 provided a poignant reminder of the inability of Internet services to scale: virtually every Internet news site was unavailable for several hours due to unprecedented demand [11]. The increasing prevalence of sophisticated denial-of-service attacks, launched simultaneously from thousands of unrelated machines, further underscores this problem.

Moreover, as our notion of Internet-based services expands to embrace a range of novel distributed systems, including global storage services [10, 20], peer-to-peer systems [6, 18], and sensor networks [5, 8], throwing more resources at the problem does not help: individual nodes in these large computing frameworks are not necessarily backed by massive data centers which can grow to meet capacity.

Despite the importance of load management, few systems directly address this problem, treating it as an issue of capacity

planning rather than preparing in advance for (inevitable) overload. Web servers, clustered middle-tier systems, databases, directory services, and file servers all require some form of overload management, though few deployed systems are architected to take this problem into account. To a large extent, this is due to inadequate interfaces for resource management. Most operating systems adhere to the principle of *resource virtualization* to simplify application development. Unfortunately, this approach makes it difficult for applications to be aware of, or adapt to, real resource limitations [24]. For example, the UNIX *malloc* interface simply returns NULL when memory cannot be allocated; an application has no way to know whether a future *malloc* operation will fail, so adapting to memory pressure is nearly impossible.

The programming models used for Internet services generally fail to express resource constraints in a meaningful way. CORBA [15], RPC [21], Java RMI [22], and now .NET [19] all expose a programming model in which distributed components communicate mainly through remote procedure call, simplifying the harnessing of remote resources through a familiar programming abstraction. Unfortunately this abstraction makes no attempt at exposing resource limits or overload conditions to the participating applications. For example, Java RMI calls can throw a generic exception due to any type of failure, but there is typically little that an RMI application can do when this occurs: should the application fail, retry the operation, or invoke an alternate interface?

This problem is compounded when Internet services are constructed through composition of many distributed systems, as is the case with the emergent field of “Web services.” Consider a Web service consisting of several independent components communicating through a common protocol such as SOAP. When one component becomes a resource bottleneck, the only overload management technique generally used is for the service to refuse additional TCP connections. While effectively shielding that service from load, other participants experience very long connection delays (e.g., due to TCP’s exponential SYN retransmit backoff behavior), causing the bottleneck to propagate through the entire distributed application.

This paper outlines a framework for building Internet services that are inherently robust to load, using two simple techniques: dynamic resource management and fine-grained admission control. While these techniques have been explored elsewhere in the

context of specific applications, we find that few Internet service programming models make them explicit. Our approach is based on a software architecture called the *staged event-driven architecture* (or SEDA), which decomposes an Internet service into a network of event-driven stages connected with explicit event queues. Load management in SEDA is accomplished by introducing a feedback loop that observes the behavior and performance of each stage, and applies resource control and admission control to effectively manage overload.

Our previous work on SEDA [25] focused primarily on the efficiency and scalability of this architecture with respect to traditional concurrent server designs. In this paper, we build on the SEDA framework by introducing adaptive overload control mechanisms and discussing the impact of overload control on the SEDA programming model. We report our experiences with several approaches for overload management in SEDA, and present initial results showing the effectiveness of these techniques in a complex Web-based email service.

2 The Need for Dynamic Overload Management

The classic approach to resource management in Internet services is static resource containment, in which *a priori* resource limits are imposed on an application or service to avoid overcommitment. Various kinds of resource limits are used: bounding the number of processes or threads within a server is a common technique, as is limiting the number of client socket connections to the service. Both of these approaches have the fundamental problem that it is generally not possible to know what the ideal resource limits should be. Setting the limit too low underutilizes resources, while setting the limit too high can lead to oversaturation and serious performance degradation under overload. Refusing to accept additional TCP connections under heavy load is inadvisable as it causes clients to retransmit the initial SYN packet with exponential backoff, leading to very long response times [25]. This approach is also too coarse-grained in the sense that even a single client can consume all of the resources in the system; imposing process or connection limits does not solve the more general resource management issue.

Another style of resource containment is that typified by a variety of real-time and multimedia systems. In this approach, resource limits are typically expressed as reservations or shares, as in “process *P* gets *X* percent of the CPU.” In this model, the operating system must be careful to account for and control the resource usage of each process. Applications are given a set of resource guarantees, and the system prevents guarantees from being exceeded through scheduling or forced termination. Though resource allocations may change over time, the system does not typically use any feedback on application performance when determining allocations. One important exception is feedback-driven scheduling [13], in which application performance is used to tune scheduling parameters.

Reservation- and share-based resource limits have been explored in depth by systems such as Scout [14], Nemesis [12], Resource Containers [3], and Cluster Reserves [2]. These techniques work well for real-time and multimedia applications, which have relatively static resource demands that can be expressed as straightforward, fixed limits. For this class of ap-

plications, guaranteeing resource availability is more important than ensuring high concurrency for a large number of varied requests in the system. Moreover, these systems are focused on resource allocation to processes or sessions, which are fairly coarse-grained entities. In an Internet service, the focus is on individual requests, for which it is permissible (and often desirable) to meet statistical performance targets over a large number of requests, rather than to enforce guarantees for particular requests.

We argue that the right approach to overload management in Internet services is feedback-driven control, in which the system actively observes its behavior and performance, and applies dynamic control to manage resources. Several systems have explored the use of dynamic overload management in Internet services. Voigt *et al.* [23] and Jamjoom [9] present approaches enabling *service differentiation* in busy Internet servers: the basic idea is to adjust the priority or admission control parameters for each class of requests to yield higher performance for more important requests. In [23], the kernel adjusts process priorities to meet per-class response time targets. When the system is overloaded, processes are blocked and eventually new connections are refused. In [9], per-class admission control is performed by traffic shaping the incoming SYN queue for new connections. The latter technique is limited to classification by client IP address, while the former rapidly accepts incoming TCP connections permitting classification by HTTP header information.

These mechanisms are approaching the kind of overload management techniques we would like to see in Internet services, yet they are inflexible in that the application itself is not designed to manage overload. Rather, overload management is provided as an OS function with generic load shedding techniques (e.g., blocking processes or rejecting connections) rather than application-specific service degradation. Also, these mechanisms are “wrapped around” existing applications rather than pushing overload control into the application design, where we argue it belongs.

3 SEDA: Making Overload Management Explicit

We have been experimenting with a new software design, the *staged event-driven architecture* (or SEDA), which is designed to provide adequate primitives for managing load in busy Internet services. In SEDA, applications are structured as a graph of event-driven *stages* connected with explicit *event queues*, as shown in Figure 1. We provide a brief overview of the architecture here; a more complete description is given in [25].

3.1 SEDA Overview

SEDA is intended to support the massive concurrency demands of large-scale Internet services, as well as to exhibit good behavior under heavy load. Traditional server designs rely on processes or threads to capture the concurrency needs of the server — a common design is to devote a thread to each client connection. However, general-purpose threads are unable to scale to the large numbers required by busy Internet services [4, 7, 16]. SEDA makes use of efficient event-driven concurrency, in which a small number of threads are used to process many simultane-

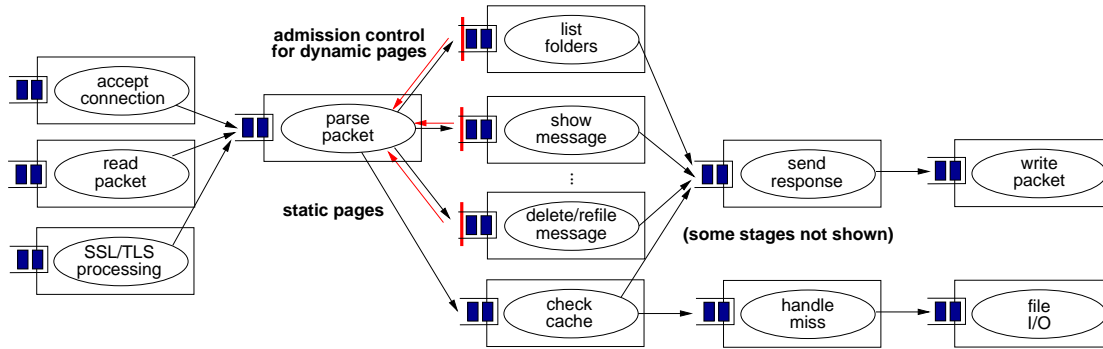


Figure 1: **Structure of the Arashi SEDA-based email service:** The service consists of a network of stages connected with explicit event queues, coupled with adaptive resource and admission control to prevent overload. For simplicity, some event paths and stages have been elided from this figure.

ous requests. This requires that request-processing operations be nonblocking, avoiding the need to devote a large number of threads to blocking I/O operations.

Event-driven server designs can often be very complex, requiring application-specific request scheduling, often resulting in labyrinthine application code. Also, the requirement the request-processing logic never block is difficult to achieve in practice, especially if legacy code is incorporated into the service. To counter the complexity of the monolithic event-driven approach, SEDA decomposes a service into a graph of stages, where each stage is internally event-driven and contains a dynamically-sized thread pool to drive its execution. This allows event processing within a stage (and across stages) to proceed in parallel, and permits application code to block for short periods of time. Additionally, the complexity of managing concurrency is significantly reduced, as each stage is responsible only for a subset of request processing, and stages are isolated from others through composition with queues.

While conceptually simple, the SEDA model has a number of desirable properties for overload management:

- **Exposure of the request stream:** Event queues make the request stream within the service explicit, allowing the application (and the underlying runtime environment) to observe and control the performance of the system, e.g., through reordering or filtering of requests.
- **Focused, application-specific admission control:** By applying fine-grained admission control to each stage, the system can avoid bottlenecks in a focused manner. For example, a stage that consumes many resources can be conditioned to load by throttling the rate at which events are admitted to just that stage, rather than refusing all new requests in a generic fashion. The application can provide its own admission control algorithms that are tailored for the particular service.
- **Performance isolation:** Requiring stages to communicate through explicit event-passing allows each stage to be insulated from others in the system for purposes of code modularity and performance isolation.

In SEDA, each stage is subject to dynamic resource control, which attempts to keep each stage within its ideal operating regime by tuning parameters of the stage’s operation. For example, one such controller adjusts the number of threads executing within each stage based on an observation of the stage’s offered load (incoming queue length) and performance (throughput). This approach frees the application programmer from manually setting “knobs” that can have a serious impact on performance. More details on resource control in SEDA are given in [25].

Each stage has an associated admission controller that guards access to the event queue for that stage. The admission controller is invoked upon each enqueue operation on a stage and may either accept or reject the given request. Numerous admission control strategies are possible, such as simple thresholding, rate limiting, or class-based prioritization. Additionally, the application may specify its own admission control policy if it has special knowledge that can drive the load conditioning decision.

When the admission controller rejects a request, the corresponding enqueue operation fails, indicating to the originating stage that there is a bottleneck in the system. Applications are therefore responsible for reacting to these “overload signals” in some way. The simplest response is to block until the downstream stage can accept the request, which leads to backpressure within the graph of stages. Another response is to drop the request, possibly sending an error message to the client or using the HTTP redirect mechanism to bounce the request to another server. More generally, SEDA applications can *degrade service* in response to overload, such as delivering lower-quality content or choosing to consume fewer resources per request. The key is that the architecture is explicit about signaling overload conditions and allows the application to participate in load management decisions.

4 Adaptive Admission Control in SEDA

The use of per-stage admission control in SEDA allows applications to be conditioned to load in a focused manner. For example, the flow of requests into stage that are the source of a resource bottleneck can be throttled. In this section we describe an adaptive, per-stage admission control technique that attempts to bound the 90th percentile response time of requests flowing

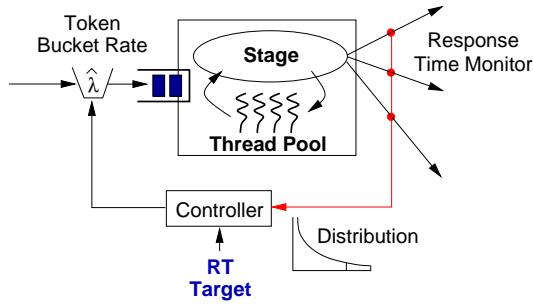


Figure 2: **Response time controller design:** The controller observes a history of response times through the stage, and adjusts the rate at which the stage accepts new requests to meet an administrator-specified 90th-percentile response time target.

through the graph of stages. We also discuss the use of service degradation and differentiation across multiple classes of requests.

4.1 Performance metrics

A variety of performance metrics have been studied in the context of overload management, including throughput and response time targets, differentiated service (e.g., the fraction of users in each class that meet a given performance target), and so forth. We focus here on *90th percentile response time* as a realistic and intuitive measure of client-perceived system performance. This metric has the benefit that it is both easy to reason about and captures administrators' (and users') intuition of Internet service performance. This is as opposed to average or maximum response time (which fail to represent the "shape" of a response time curve), or throughput (which depends greatly on the user's connection to the service and has little relationship with user-perceived performance).

In this context, the system administrator would specify a target value for the 90th percentile response time exhibited by requests flowing through the service. The target value may be parameterized by relative utility of each request, for example, based on request type or user classification. An example might be to specify a lower response time target for requests from users with more items in their shopping cart. Our current implementation allows separate response time targets to be specified for each stage in the service, as well as for different classes of users (based on IP address, request header information, or HTTP cookies).

4.2 Response time controller design

The design of the per-stage overload controller in SEDA is shown in Figure 2. The controller consists of several components. A *monitor* measures response times for each request passing through a stage. The measured 90th percentile response time over some interval is passed to the *controller* which adjusts the *admission control parameters* based on the administrator-supplied response-time *target*. In the current design, the controller adjusts the rate at which new requests are admitted into the stage's queue by adjusting the rate at which new tokens are generated in a token bucket traffic shaper.

Due to space limitations we present a brief overview of the overload controller implementation. The basic overload control algorithm makes use of additive-increase/multiplicative-decrease tuning of the token bucket rate based on the current observation of the 90th percentile response time. The overload controller is implemented as a function invoked by the stage's event-processing thread after some number of requests has been processed. This implies that the overload controller will not run when the token bucket rate is low; the algorithm therefore "times out" and performs a recalculation of the 90th percentile response time estimate after a certain interval. When the 90th percentile response time estimate is above a high-water mark (e.g., 10% above the administrator-specified target), the token bucket rate is reduced by a multiplicative factor (e.g., dividing the admission rate by 2). When the estimate is below a low-water mark, the token bucket rate is increased by a small additive factor. The rate increase is proportional to the difference between the current response time estimate and the target; a larger error leads to a greater increase in the admission rate.

4.3 Service degradation and differentiation

In addition to the basic response-time controller, we have implemented mechanisms permitting applications to degrade service under load, as well as to provide differentiated levels of service based on the type of request or user class. A complete discussion of these mechanisms is beyond the scope of this paper, though we touch on them briefly here.

Rather than rejecting requests, SEDA applications may degrade the quality of delivered service in order to admit a larger number of requests given a response-time target. SEDA itself does not implement service degradation mechanisms, but rather signals overload to applications in a way that allows them to degrade if possible. SEDA allows application code to obtain the current 90th percentile response time estimate from the overload controller, as well as to enable or disable the admission control mechanism for a given stage. This allows an application to implement degradation by periodically sampling the current response time estimate and comparing it to the administrator-specified target. If service degradation is ineffective (say, because the load is too high to support even the lowest quality setting), the stage can re-enable admission control to cause requests to be rejected.

Likewise, by prioritizing requests from certain users over others, a SEDA application can implement various policies related to class-based service level agreements. A common example is to give better service to requests from "gold" customers (who might pay more money for the offered service). Building on the basic response time controller described above, We have implemented service differentiation in SEDA using a controller that more aggressively rejects lower-class requests than higher-class requests when a stage is overloaded.

5 Evaluation

In this section we evaluate the overload controllers using two SEDA applications: a Web-based interface to email, and a Web server benchmark that is able to degrade service under heavy load.

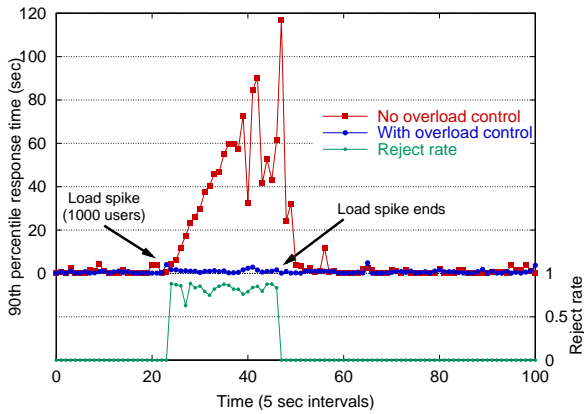


Figure 3: Overload control under a massive load spike: This figure shows the 90th percentile response time experienced by clients using the Arashi e-mail service under a massive load spike (from 10 users to 1000 users). Without overload control, response times grow without bound; with overload control (using a 90th percentile response time target of 1 second), there is a small increase during load but response times quickly stabilize. The lower portion of the figure shows the fraction of requests rejected by the overload controller.

5.1 Arashi: A SEDA-based e-mail service

Arashi is a complex, Web-based email application that is akin to Hotmail or Yahoo! Mail, allowing users to access email through a Web browser interface with various functions: managing email folders, deleting and refiling messages, searching for messages, and so forth. Arashi (shown in Figure 1) is built as a SEDA application consisting of 16 stages, each stage handling some aspect of request processing such as HTTP parsing, disk I/O, or dynamic page generation. Email is stored in a MySQL database which runs on the same machine as the Arashi SEDA service; in this way, Arashi's admission control mechanisms effectively condition load on the database. Simulated clients generate load against the Arashi service using a realistic request distribution based on traces from the Berkeley departmental IMAP server.

Response-time-driven overload control is applied to each of the six stages that perform dynamic page processing. These stages are the bottleneck in the system as they perform database access and HTML page generation; the other stages are relatively lightweight. Each stage corresponds to one type of user request (login, listing folders, listing messages, showing a message, deleting/refiling messages and folders, and searching message headers). When the admission controller rejects a request, the HTTP processing stage sends an error message to the client indicating that the service is busy. The client records the error and waits for 5 seconds before attempting to login to the service again.

Figure 3 shows the performance of the overload controller under a massive load spike on the Arashi e-mail service. A base load of 10 users is rapidly accessing the service when a “flash crowd” of 1000 additional users arrive. Without overload control, client-measured response times grow to be very large. The overload controller maintains a 90th percentile response time target of 1 second, rejecting about 70% to 80% of requests dur-

ing the spike. Without overload control, there is an enormous increase in response times during the load spike, making the service effectively unusable for all users.

This is in contrast to the common approach of limiting the number of client TCP connections to the service, which does not actively monitor response times (a small number of clients could cause a large response time spike), nor give users any indication of overload. In fact, refusing TCP connections has a negative impact on user-perceived response time, as the client's TCP stack transparently retries connection requests with exponential backoff.

We claim that giving 20% of the users good service and 80% of the users some indication that the site is overloaded is better than giving *all* users unacceptable service. However, this comes down to a question of what policy a service wants to adopt for managing heavy load. Recall once more that the service need not reject requests outright — it could redirect them to another server, degrade service, or perform an alternate action. The SEDA design allows a wide range of policies to be implemented, using per-stage admission control as a load management primitive.

5.2 Service degradation experiments

As discussed previously, SEDA applications can respond to overload by degrading the fidelity of the service offered to clients. This technique can be combined with admission control, for example, by rejecting requests when the lowest service quality still leads to overload.

To demonstrate the use of service degradation in SEDA, we use a simple Web server that responds to each request with a dynamically-generated HTML page that requires significant resources to generate. A single stage acts as a bottleneck in this service; for each request, the stage reads a varying amount of data from a file, computes checksums on the file data, and produces a dynamically-generated HTML page in response. The stage has an associated *quality factor* that controls the amount of data read from the file and the number of checksums computed. By reducing the quality factor, the stage consumes fewer resources, but provides “lower quality” service to clients.

Using the overload control interfaces in SEDA, the stage monitors its own 90th percentile response time and reduces the quality factor when it is over the administrator-specified limit. Likewise, the quality factor is increased slowly when the response time is below the limit. Service degradation may be performed either independently or in conjunction with the response-time admission controller described above. If degradation is used alone, then under overload all clients are given service but at a reduced quality level. In extreme cases, however, the lowest quality setting may still lead to very large response times. The stage may optionally re-enable the admission controller when the quality factor is at its lowest setting and response times continue to exceed the target.

Figure 4 shows the effect of service degradation under an extreme load spike, both with and without the aid of admission control. As the figure shows, service degradation alone does a fair job of managing overload, though re-enabling the admission controller under heavy load is much more effective. Note that

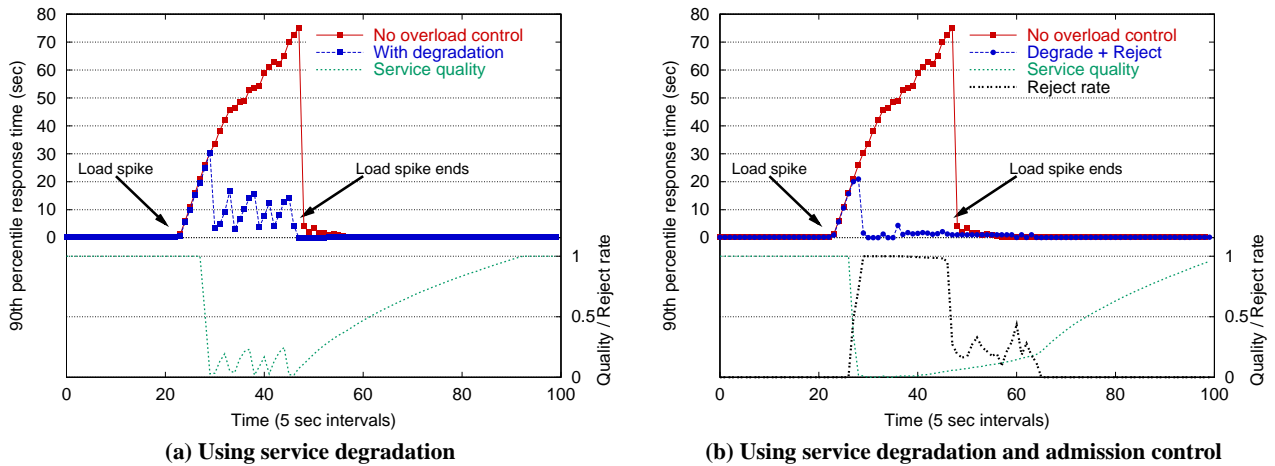


Figure 4: Effect of service degradation vs. admission control: This figure shows the 90th percentile response time experienced by clients accessing a simple service consisting of a single bottleneck stage. The stage is capable of reducing the quality of service delivered to clients in order to meet response time demands. Here, the 90th percentile response time target is set to 5 seconds. With no degradation or admission control, response times grow large under a massive load spike of 1000 users. Service degradation alone does a fair job of meeting the response time target under overload, though service degradation coupled with admission control is much more effective.

when admission control is used, a very large fraction (99%) of the requests are rejected; this is due to the extreme nature of the load spike and the inability of the bottleneck stage to meet the performance target, even at a degraded level of service.

6 Conclusions and Future Directions

This position paper has argued that it is critically important to address the problem of overload from an Internet service design perspective, rather than through *ad hoc* approaches lashed onto existing systems. Rather than static resource partitioning or prioritization, we claim that the right way to approach overload management is to use feedback and dynamic control. This approach is more flexible, less prone to underutilization of resources, and avoids the use of static “knobs” that can be difficult for a system administrator to tune. In our approach, the administrator specifies only high-level performance targets which are met by feedback-driven controllers.

We also argue that it is necessary to expose overload to the application, rather than hiding load management decisions in an underlying OS or runtime system. Application awareness of overload conditions allows the service to make informed resource management decisions, such as degrading the quality of service. In the SEDA model, overload is exposed to applications through explicit signals in the form of cross-stage enqueue failures. Our initial results with this design, as well as considerable scalability and robustness measurements presented elsewhere [25], support the claim that the SEDA approach is an effective way to build robust Internet services.

A wide range of open questions remain in the Internet service design space. We feel that the most important issues have to do with robustness and management of heavy load, rather than raw performance, which has been much of the research community’s focus up to this point. Some of the interesting research challenges raised by the SEDA design are outlined below.

User-level versus kernel-level load management: An interesting aspect of SEDA is that it is purely a “user level” mechanism, acting as a resource management middleware sitting between applications and the underlying operating system. However, if given the opportunity to design an OS for scalable Internet services, many interesting ideas could be investigated, such as scalable I/O primitives, SEDA-aware thread scheduling, and application-specific resource management.

Design and tuning of control mechanisms: Introducing feedback as a mechanism for overload control raises a number of questions. For example, how should controller parameters be tuned? We have relied mainly on a heuristic approach to controller design, though more formal, control-theoretic techniques are possible [17]. Control theory provides a valuable framework for designing and evaluating feedback-driven systems, though many of the traditional techniques rely upon good mathematical models of system behavior, which are often unavailable for complex software systems. The interaction between multiple levels of control in the system — for example, the interplay between queue admission control and tuning per-stage thread pool sizes — is also largely unexplored.

Composition rules for service components: The presence of an expressive interface for overload management raises the larger question of how to use this mechanism across large applications. Within a SEDA application, stages may independently shed load by rejecting enqueue operations; stages must therefore be defensive with respect to generating load for downstream stages. An interesting design issue arises with respect to composing stages each with their own load-shedding policy. Can any two stages be directly composed, or do some forms of overload control preclude direct communication between stages? Further research might yield a set of composition rules that mandate the interactions between admission-controlled service components.

Overload management as part of the Internet infrastructure: Finally, we hope that future distributed systems developers consider overload as an important design consideration, rather than as only a question of resource provisioning. While current distributed programming models, such as RPC, fail to expose overload, novel application domains such as overlay networks [1] and peer-to-peer systems [10] have the opportunity to rethink this approach. Doing so will hopefully lead to an Internet infrastructure that is more robust to extreme variance of demand.

References

- [1] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, Banff, Canada, October 2001.
- [2] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, June 2000.
- [3] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, February 1999.
- [4] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proc. USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.
- [5] A. Cerpa and D. Estrin. ASCENT: Adaptive self-configuring sensor networks topologies. In *Proceedings of the Twenty First International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, New York, NY, June 2002.
- [6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP-18)*, Chateau Lake Louise, Canada, October 2001.
- [7] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proc. Fourth Symposium on Operating System Design and Implementation (OSDI 2000)*, October 2000.
- [8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *Proceedings of ASPLOS 2000*, Cambridge, MA, November 2000.
- [9] H. Jamjoom and J. Reumann. QGuard: Protecting Internet servers from overload. Technical Report CSE-TR-427-00, University of Michigan Department of Computer Science and Engineering, 2000.
- [10] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.
- [11] W. LeFebvre. CNN.com: Facing a world crisis. Invited talk at LISA'01, December 2001.
- [12] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14:1280–1297, September 1996.
- [13] C. Lu, J. Stankovic, G. Tao, and S. Son. Design and evaluation of a feedback control EDF algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1999.
- [14] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proc. OSDI '96*, October 1996.
- [15] Open Management Group. The Common Object Request Broker: Architecture and specification, revision 2.3, June 1999.
- [16] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proc. 1999 Annual Usenix Technical Conference*, June 1999.
- [17] S. Parekh, N. Gandhi, J. L. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *Proc. IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001.
- [18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM 2001*, San Diego, CA, August 2001.
- [19] J. Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.
- [20] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP-18)*, Chateau Lake Louise, Canada, October 2001.
- [21] Sun Microsystems. RPC: Remote Procedure Call Protocol Specification Version 2. Internet Network Working Group RFC1057, June 1988.
- [22] Sun Microsystems, Inc. Java Remote Method Invocation. <http://java.sun.com/products/jdk/rmi/>.
- [23] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [24] M. Welsh and D. Culler. Virtualization considered harmful: OS design directions for well-conditioned services. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*.
- [25] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP-18)*, Chateau Lake Louise, Canada, October 2001.

Rewind, Repair, Replay: Three R's to Dependability

Aaron B. Brown and David A. Patterson

University of California at Berkeley, EECS Computer Science Division

387 Soda Hall #1776, Berkeley, CA, 94720-1776, USA

{abrown, patterson}@cs.berkeley.edu

Abstract

Motivated by the growth of web and infrastructure services and their susceptibility to human operator-related failures, we introduce system-level undo as a recovery mechanism designed to improve service dependability. Undo enables system operators to recover from their inevitable mistakes and furthermore enables retroactive repair of problems that were not fixed quickly enough to prevent detrimental effects. We present the “three R’s”, a model of undo that matches the needs of human error recovery and retroactive repair; discuss several of the issues raised by this undo model; and introduce an initial architectural framework for undoable systems using the example of an undoable e-mail service system.

1. Introduction

The need for a dependable computing infrastructure has never been more urgent. The world is shifting to a model where data is stored and maintained in centralized servers and doled out to clients via network services; we have seen the beginnings of this trend over the last few years with the growth of Internet-based services, portals, and e-commerce. As the trend accelerates further with the deployment of technologies such as pervasive wireless networking, mobile devices, .NET, and J2EE, the social and financial impact of dependability problems in the infrastructure promises to be enormous.

One of the primary impediments to infrastructure dependability today is the human operator (a.k.a. system administrator). Human operator error is the leading cause of outages across a spectrum of systems ranging from Internet services to the US telephone network [2] [7] [14]. When operators don't create outages, they often compound them by not responding quickly enough to fix the problems before damage is done.

What are we to do? One option is to eliminate the human operator entirely. This may work for small embedded devices, but it doesn't apply to the large systems with hard state that make up the network service infrastructure of the future. Furthermore, studies from psychology and system accident theory leave little room for debate:

attempts to automate away human operators in large systems invariably fail due the *automation irony*¹ [15].

The only viable alternative, then, is to build infrastructure systems that accept and compensate for the inevitable weaknesses of their human operators. Future systems should recover easily from operator mistakes, give the operator an environment in which trial-and-error reasoning is possible, and harness the unique human capacity for hindsight by allowing *retroactive* repairs once problems have been manifested. There is a recovery mechanism that has these properties, and it is one that we use every day in our word processors and spreadsheets: *undo*. Unfortunately, undo as a recovery model has been limited to the application level, where it is insufficient to tackle the operational problems that plague infrastructure systems: operator errors made during upgrades and reconfiguration, external virus and hacker attacks, and unanticipated problems detected too late for their effects to be contained.

To address these problems, we introduce *system-level undo*, an undo-based recovery model that covers all levels of the system, not just the application. The crux of our undo model is that it disambiguates user intentions from their manifestations in system state, allowing the undo mechanism to repair problems in system state without losing user data.

Although the technological underpinnings of our undo mechanism are simple—a combination of non-overwriting storage and logging of user inputs—the policy choices in an undo implementation are complex. Most challenging is dealing with the problem of *external inconsistency*, where the undo process alters or revokes erroneous state changes that have already been seen by external end-users. Another challenge lies in identifying and tracking state that should be made recoverable through undo: a good undo mechanism will preserve user data while allowing arbitrary

¹ The automation irony captures two problems with automation. First, automation shifts the burden of correctness from the operator to the designer, requiring that the designer anticipate and correctly address all possible failure scenarios. Second, as the designer is rarely perfect, automated systems almost always can reach exceptional states that require human intervention. These exceptional states correspond to the most challenging, obscure problems; psychological studies routinely show that humans are most prone to mistakes on these types of problems, especially when automation has eliminated normal day-to-day interaction with the system.

repairs to system state, and drawing the dividing line between recoverable and non-recoverable changes is non-trivial. A final challenge is in constructing an undo system that works at multiple granularities: cluster-wide, per-system, and per-user.

In the remainder of this paper, we expand on these challenges, identifying possible solutions and integrating them into an architectural framework for undo-capable systems. We begin in Section 2 by defining the essence of the undo process: the “Three R’s” of rewind, repair, and replay. Section 3 puts the Three R’s into the context of related work on undo systems. Section 4 delves deeper into the issues and challenges raised by the Three R’s model, and Section 5 introduces an architecture for Three R’s-undo that addresses these challenges. Finally, we wrap up with conclusions and future work in Section 6.

2. The Three R’s: an Undo Model Akin to Time Travel

To support retroactive repair and recovery from operator error, we propose an undo model based on a 3-step process that we call “the three R’s”: *Rewind*, *Repair*, and *Replay*. In the *rewind* step, all system state (including user data as well as OS and application hard state) is reverted to its contents at an earlier time (before the error occurred). In the *repair* step, the operator is allowed to make any changes to the system he or she wants. Changes could include fixing a latent error, installing a preventative filter or patch, retrying an operation that was unsuccessful the first time around (like a software upgrade of the application or OS), or even simply omitting an action that caused problems (like accidental deletion of important data). Finally, in the *replay* step, the undo system re-executes all *end-user interactions* with the system, allowing them to be reprocessed with the changes made during the repair step.

A convenient metaphor for understanding the 3R undo model is to think of it as time travel. In a common portrayal of time travel in science-fiction, a protagonist travels back through time to right a wrong. By making changes to the timeline in a past time frame, the protagonist fixes problems and averts disaster, and the effects of those changes are instantaneously propagated forward to the present. The 3R undo model offers a similar sequence of events. The rewind step is the equivalent of traveling back in time, in this case to the point in time before an error occurred. The repair step is equivalent to changing the timeline: the course of events is altered such that the error is repaired or avoided. Finally, the replay step propagates the effects of the repair forward to the present by reexecuting—in the context of the repaired system—all events in the timeline between the repairs and the present. Since the events are replayed in the context of the repaired system, they reflect the effects of the repairs and any

incorrect behavior resulting from the original error is cancelled out.

All three steps in the 3R model are required to achieve effective retroactive repair and recovery from operator error. Without rewind, recovery would not be possible since the state changes induced by the error could not be revoked. Without repair, the error itself could not be corrected. Without replay, all user interactions and updates between the rewind point and the present would be lost.

3. Related Work

Our 3R undo model draws on a long heritage of work in temporal recovery and undoable systems. Probably the most ground-breaking work to date on sophisticated undo systems can be found in Edwards’s Timewarp system [4], a framework for building collaborative productivity applications that supports a rich and malleable view of time and history. In Timewarp, different users work autonomously on shared state, each generating explicitly-visible histories of actions over that state; users can rewind their state, alter their histories, and replay changes at will. Timewarp also defines a framework for detecting and managing undo-induced inconsistencies based on an analysis of the static relationships between user actions [3].

Much of our 3R undo model is similar to Timewarp. In particular, both systems are based on the command-object idiom [9], where history is tracked in terms of user operations representing instances of generic actions. However, 3R undo is targeted at a much lower level of the system and hence has some key differences. First, 3R undo is designed to undo and replay entire systems—from the OS up—and not just application-level events; this requires that 3R rewind be implemented physically whereas Timewarp rollback is done logically. Note that physical rewind is also necessary in 3R undo to cope with buggy systems where user actions can have potentially-arbitrary effects on state.

More fundamentally, 3R undo makes no assumptions about the structure of repair, allowing arbitrary changes to the system itself during the repair phase. In contrast, Timewarp repairs are limited to inserting or deleting well-known actions from the history.

Finally, 3R undo and Timewarp take different approaches to the problem of undo-induced inconsistencies. Timewarp only manages inconsistencies arising as the result of conflicts between two or more operations in the system’s history, meaning that it assumes the behavior of operations is repeatable and well-known. Because of its support for unconstrained repair, 3R undo cannot make that assumption, and hence models inconsistencies as arising from conflicts between individual operations and the system state context in which they are executed. The 3R approach permits much more flexibility in repair but sacri-

fices the ability to perform extensive static analysis of inconsistency; an interesting area of future work is to merge the two approaches to provide static analysis of expected behavior while dynamically handling worst-case behavior.

Besides Timewarp, there are several other influential systems that support time-travel-like undo. Freeman and Gelernter's Lifestreams is a system that explicitly manages a user's state repository as a temporal stream of documents [8]; Rekimoto's Time-Machine Computing is a similar system that merges temporal and spatial representations of history [16]. Unlike 3R undo, these systems only support a linear, unchanging view of history without repair: users can observe the past and even add events to the future timeline, but changes to the past are limited to simple annotations that have no side effects on future state. Roxio's GoBack utility is a commercial product providing similar utility to the users of Windows environments [17]. Several graphical-editing environments define models that support editing of past history (with Kurlander's Chimera being one of the most influential [11]), but unlike 3R undo or Timewarp, none of these systems address management of undo-induced inconsistencies.

Finally, in terms of implementation, many systems offer a subset of the 3R properties but none offers full 3R semantics at the system level. For example, backup/restore or checkpointing schemes [1] [6] [12] offer rewind/repair or rewind/replay, but deny the ability to roll forward once changes have been made. Recovery systems for transactional relational databases use rewind/replay to recover from crashes, deadlocks, and other fatal events [13], but again do not offer the ability to interject repair into the recovery cycle; the standard transaction model does not allow committed transactions to be altered or removed. Some extended transaction models do allow altering or undoing of committed transactions (a good example is Korth et al.'s model of compensating transactions, upon which our compensation approach in Section 4.2 is loosely based [10]), but these models are primarily theoretical and, like Timewarp, are based on operation-operation conflicts rather than state-operation conflicts.

In summary, our 3R undo model offers a unique combination of properties not found together in any existing undo system of which we are aware. It operates at a low level of the system, allowing recovery from problems affecting the operating system and higher software levels, supports unconstrained repair with forward-propagation of changes, and allows detection and management of repair-induced inconsistencies.

4. Challenges in the 3R Undo Model

4.1. Tracking recoverable state for replay

When an undo is carried out under the 3R undo model, all state changes made since the undo point are wiped out during the rewind step. It is the responsibility of the replay step to restore all state changes that are important to the end user. Defining exactly what state this encompasses is tricky, especially when repairs could radically change the physical representation of state (*e.g.*, an upgrade of a mail server that rewrites the on-disk mailbox format). Ideally, the replay mechanism should track and preserve end-user *intent* rather than specific state changes. For example, in an undoable e-mail system, a user's act of deleting a message should be recorded as "delete message with Msg-ID x ", not "alter byte range $m - n$ in file z ". By tracking user updates at an intentional level, the replay system has the best hope of preserving the state that the user cares about while leaving as much flexibility for repair as possible.

In the network service environment that we are targeting, users interact with the system through standardized application protocols, so the easiest way to achieve intentional tracking of user updates is to intercept and record user interactions at the protocol level. Most network service protocols define a set of verbs that the client can use to express desired actions, and most are designed so that state is referenced using logical names divorced from any particular internal state representation. Protocols also have the advantage of being reasonably standard and well-defined. Good examples include SMTP and IMAP for email, JDBC/SQL for databases, and XML/SOAP for the emerging online application frameworks. Tracking interactions at the level of protocol verbs and logical state names automatically provides a record of user intent that is independent of the details of the application itself; in fact, it should be possible to completely swap out one server implementation for another during the repair phase and still be able to replay user interactions, as the protocol itself is unlikely to change across implementations.

Finally, up to now we have defined replay as only affecting user state, but have ignored the issue of whether repairs are tracked. As with user updates, to track and replay repairs the undo system would have to log the intent of the repairs, not their effects on state. While this is feasible for protocol-limited user interactions, it becomes a nightmare when the set of possible changes is limited only by the operator's human ingenuity, not a list of protocol commands. Thus for practical reasons we make the choice to not allow replay of repairs in our undo model; we may explore the possibility in future work. Note that this restriction does have implications for the undo history paradigm: the operator can use undo to back up over arbitrary repairs and changes (and in a simple extension can

immediately undo the undo before making any changes), but once changes are made in the repair phase, only user state will be restored on replay.

4.2. External inconsistency

A favorite device of time-travel fiction is the time-paradox, where alterations to the past timeline effect unexpected changes in the present. In these paradoxes, the time-traveling protagonist, whose memories are typically isolated from the altered timeline, sees the “new” present as inconsistent.

The same problem plagues system-level undo: during the undo cycle, repairs change the past state of the system, and replay propagates those changes forward to produce a new version of the present that is likely inconsistent with the view of the present seen before the undo cycle. For example, in an email system, a retroactive repair could consist of installing a spam- or virus-blocking filter. When replayed forward, formerly-delivered mail messages might be squashed by the new filter. A user who had read, forwarded, or replied to those messages would see the system as inconsistent with his or her expectations once the undo cycle was complete. Note that this problem of post-undo external inconsistency arises only when state that has formerly been made visible to an external entity (*i.e.*, the user) is altered by the undo cycle; state that has not been externalized cannot cause inconsistencies.

As with the similar output commit problem discussed in the checkpointing literature [6], there is no complete fix for the external inconsistency problem; possible solutions involve managing the inconsistency rather than eliminating it. The easiest solution is to simply ignore the inconsistency, assuming that the user will tolerate it. This approach is best suited for minor inconsistencies in applications with relaxed semantics, for example when the inconsistency causes reordering of message delivery in an email system or changes item availability estimates in an e-commerce system. When the inconsistency is too large to ignore, the best solution is to use compensating or explanatory actions to help the user adjust to it. For example, in our email scenario above, we could replace the removed message in the user’s mailbox with a new message explaining why the original message was removed; this technique is used effectively today by virus-scanning email gateways.

When the entity that externalizes state is not an end-user but another computer system, there are more powerful solutions available. One, which removes inconsistencies entirely, is to expand the boundary of the undo system to encompass the external system. This can be done by propagating undo requests across system boundaries so that when externalized state is changed the external system is rolled back and replayed with the new version of the

externalized state. This approach must be used with care, as the boundary may have to be drawn arbitrarily large to completely tolerate the inconsistency; however, a small increase in boundary size may reduce the inconsistency to the point where it can be tolerated by a human user. Another approach when the externalizer is a computer is to delay the execution of externalizing actions for a given time period; during this undo window, the actions can be rolled back and altered without inconsistency. This approach is limited to cases where the actions are asynchronous and not time-critical, like delivering email to an external system or generating bounce messages upon a delivery failure.

4.3. Granularity of undo

To be most useful, undo as a recovery mechanism should be available at multiple granularities. A user might want to use undo to recover from a mistake affecting only his or her state; it should not be necessary to rewind/replay the entire system in order for this to happen. Conversely, the system operator must still be able to apply undo across all system state in order to recover from system-wide failure or to carry out low-level repairs that affect all users. An extension of this problem occurs in a clustered system, where it would be useful from an efficiency standpoint to support undo on the per-node level as well as the cross-cluster level.

Exposing undo at multiple granularities raises some challenges, most significantly in managing and coordinating the timelines of state at different levels of the system. For example, if a user in an email system has rewound his or her own mailbox, and the system operator then wants to rewind the entire system, a policy is needed to determine which rewind request takes precedence, and coordination is necessary to ensure that all state ends up at the correct point in time upon replay.

A further challenge arises when implementation is considered: to support fine-grained undo at the per-user level, system state must be divided into per-user state and shared state and dependencies between the two types must be respected on rewind and replay; similar issues apply to the logs of user actions used for replay.

5. An Architecture for 3R-Undo

To explore and address the challenges laid out above, we have been developing an architecture and implementation of a 3R-Undo layer. Our initial application of undo is in an e-mail service system, chosen as it is an essential service in today’s Internet environment, but one of our goals is to make the implementation as generic as possible so that it can be easily adapted to other applications.

5.1. Basic structure

Our architecture begins by defining the basic structure of an undoable system. Following the discussion in Section 4.1, we require a verb-based application interface that uses logical state names; this allows the undo system to properly disambiguate recoverable user state changes from other system events. The verb-based interface should cover all interactions with the system that affect user-visible state, including operator interfaces that are used to create, delete, move, and modify user state repositories. If existing protocols satisfy these requirements, then all is well; otherwise, new protocols can be created and layered above the existing interfaces. In our target case of e-mail, the IMAP and SMTP protocols define most of what is needed in a verb-based interface for the end-user, although they need a small extension to the naming semantics in order to provide globally-unique and time-invariant state names. Also required is a new verb-based protocol to standardize and encapsulate the parts of the operator interface affecting user state repositories (such as accounts or mail-drops), which we are attempting to develop.

A consequence of the verb-based interface requirement is that undo functionality is best implemented as a wrapper layer surrounding the application itself, interposing on the interface to track and replay state-changing events. Keeping the undo layer outside of the application itself leaves the most flexibility for repair and minimizes the potential for introducing new bugs into the undo system as the application evolves. As a fringe benefit, it also enables straightforward geographic replication of undo-wrapped services, as the high-level intentional undo log can be shipped from one instance of the service to another and applied using the same replay mechanisms used for an undo cycle. We will assume this wrapper-based architecture, shown diagrammatically in Figure 1, for the remainder of this paper.

5.2. Managing external inconsistency

Probably the most difficult challenge in architecting undo is in developing a generic mechanism to detect and manage external inconsistency. Our approach is based on an analysis of the history recorded by the undo wrapper: we augment the log with enough information to allow the undo system to track inter-verb dependencies and identify unsafe operations that result in the externalization of inconsistent state. Once identified, these inconsistencies can be addressed using one of the possible solutions introduced in Section 4.2: applying compensating actions, extending the boundary of undo, or ignoring the inconsistencies entirely. A key property of our approach is that it uses a declarative specification of the inter-verb relationships, allowing the system designer to identify and

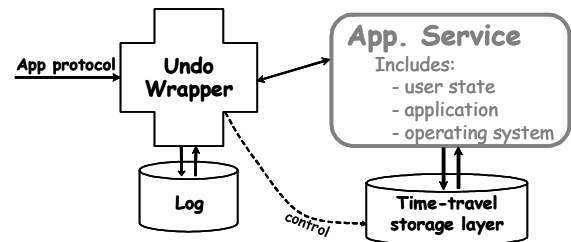


Figure 1: Basic undo architecture. The application service is wrapped with an undo layer that interposes on the verb-based application protocol, logging the intent of user changes for later replay. The undo wrapper also manages a time-travel storage layer used to implement rewind, and tracks state externalization to manage undo-induced inconsistency.

exhaustively test all possible inconsistency conditions; this provides increased confidence in the undo system's dependability, and provides the tools needed to verify *a priori* that all possible undo scenarios can be handled.

The crux of our approach to external inconsistency is an understanding of undo history. We define a *history* as the sequence of *operations* recorded/logged by the undo wrapper, where an operation is an instance of one of the *verbs* in the application interface. In our e-mail application, a verb might be to *FETCH* a message from the mail store, a corresponding operation would be a particular user's fetch of a particular message at a particular point in time, and the operation would appear alongside other operations in a history of all user interactions with the mail system.

The potential for external inconsistency occurs during the replay stage of undo when an *unsafe* operation appears in the history. Unsafe operations are operations that produce different results during replay than they did during their original execution. To detect unsafe operations, we augment our logging of operations with the following fields (note that these can be specified to the undo system declaratively in template form along with the interface verbs, allowing a generic undo layer to manage the details of operation logging):

- the names of the set of state entities needed to carry out the operation
- a set of preconditions over those state entities that must be satisfied for the verb to be successfully re-executed and to produce acceptably consistent results with the original execution.

The preconditions are generated dynamically as the operation is logged, and incorporate whatever tests are necessary to guarantee acceptable consistency. For example, in our e-mail scenario, the fetch operation might be logged with a precondition that checks a hash of the message body against a hash taken when the original operation

was logged; a discrepancy indicates that the message contents have been altered by repair. Another e-mail example is mail delivery to a folder. The logged operation might include preconditions to check for the existence and read-write status of the mail folder; if during replay the folder was missing or found to be read-only, the operation becomes unsafe.

Careful specification of preconditions is essential if unsafe operations are to be properly identified. If specified too broadly, the undo system will miss unsafe operations and allow inconsistencies to propagate without compensation; if specified too narrowly, there is the risk that perfectly acceptable replay-induced inconsistencies will be flagged as unsafe, hence limiting the transparency of repairs. Specifying the appropriate preconditions is probably the most challenging part of designing an undoable system.

Preconditions and required-state lists give the undo system the information needed to detect unsafe operations during replay, but further mechanism is needed to determine if those unsafe actions produce external inconsistency. Recall that a main purpose of an undo mechanism is to *allow* changes and repairs to the system, since those changes could be converting previously-erroneous results to correct results. Thus unsafe actions are actually desirable, and they only cause the problem of external inconsistency if the state alterations they make are externalized later in the history.

We therefore return to the importance of history: the level of inconsistency is a property of the history, not of individual operations. We define three classes of history with regard to the level of external inconsistency:

- *replay-safe*: the history can be re-executed as is without causing visible external inconsistency
- *replay-acceptable*: re-execution of the history causes visible but acceptable external inconsistency (for example, the history includes compensating actions)
- *replay-unsafe*: the history cannot be re-executed without causing unacceptable external inconsistency.

Without repair, any unedited history recorded by the undo system will be replay-safe. But once we introduce repair, replay-unsafe histories can arise as unsafe operations appear and are externalized later in the history. Furthermore, repairs can consist of alterations to the history itself, with operations deleted, added, or changed; if these changes affect later-externalized state, the history again becomes unsafe.

The undo system must detect when history is replay-unsafe, as these histories need to be converted to at least replay-acceptability for the undo system to work. This is done by tracing dependencies from unsafe operations to externalizing operations, an analysis enabled by again

augmenting the undo-logging of operations with the following fields (which also can be generically specified in template form):

- an indication of which of the state entities are expected to be modified by the verb's execution
- an indication of which of the state entities are externalized by the verb's execution
- the amount of time that the results of the verb's execution are delayed before being externalized (to capture scenarios where inconsistency is avoided by delaying notification of asynchronous actions).

Given an unsafe operation that "taints" certain state entities (*i.e.*, where preconditions are violated or where the state is modified by the unsafe operation), the undo system can walk the history forward, propagating the "tainted" status to operations that use it as input, and so on. If the undo system finds an operation that externalizes the tainted state, then the potential external inconsistency represented by the original unsafe operation is realized. Knowing this, the undo system can make a decision of how to manage the inconsistency, for example by rewriting the history to insert compensation around the original unsafe operation.

As a concrete illustration of how the detection and compensation process might work, consider the following scenario, which begins when a message containing a virus is delivered to a user's inbox. Before opening the message contents, the user copies the message into a folder, then moves the copy into a second folder. At this time, the system operator realizes that the system is being attacked by a mail virus, and invokes undo. After rolling time back to before the point where our user's message was delivered, the operator installs a filter that discards virus-laden messages, then invokes replay.

During replay, the undo system first executes the deliver operation, which, because of the filter, now discards the message. When replay reaches the copy operation, its preconditions are violated because the message in question was never delivered, and hence it is an unsafe operation. Furthermore, the copy operation externalizes the existence of the message, so the history is replay-unsafe and a compensating action is needed, perhaps inserting a placeholder message into the user's inbox with an explanation for the missing original message. With this compensation, the original replay-unsafe history has been transformed into replay-acceptability, and thus the now-safe copy operation can be executed (on the placeholder message), as can the subsequent move operation. A variant on this scenario would be a system where the operator is willing to accept greater inconsistency by not considering the copy operation as externalizing the message; in this case, the entire history would be replay-safe since there is

no externalizing operation that depends on the unsafe deliver operation.

5.3. Supporting multiple-granularity undo

So far, we have laid out an undo system architecture that addresses two of the three major challenges introduced in Section 4: tracking recoverable state and managing external inconsistency. While at this point we have not yet developed a general solution to the remaining challenge of multiple-granularity undo, we believe that the architecture developed in the previous sections may offer many of the tools needed to analyze and solve the multiple-granularity problem. In particular, the same mechanisms used for detecting unsafe actions, tracing dependencies, and counteracting inconsistency can be used to manage undo dependencies that cross granules of system state.

For example, consider a system supporting per-user undo and a scenario where user *A* wants to replay an operation that updates state from user *B* (or globally-shared state). The replay system can detect this using the dependency information in the augmented undo log, and, depending on the system policy, can either initiate a cascading undo of user *B*'s state or can treat the operation as unsafe and apply the appropriate compensations to *A*'s state. To support such analyses, the architecture needs only the extensions of divvying up system state into independently-undoable per-user state repositories and of logically splitting the system history into per-user histories; the approach to multi-level undo used by Edwards et al. in the Flatland system [5] may prove useful here, although it will need significant adaptation to remove the assumption that operation side-effects are tracked in the history.

5.4. Implications for applications

As we have defined it, our undo architecture imposes some constraints on the applications that can be wrapped with undo functionality. We can treat these constraints as essentially defining a template for undoable applications; besides helping to identify existing applications/services that can be easily extended with undo, the template provides implementors with a guide to constructing new undoable services.

The template for an undo-wrappable service is characterized by a set of properties that must be met for our architecture to be applicable:

- Clients must access the service through well-defined, narrow verb-based interfaces that identify state with logical names; the undo wrapper imposes on this interface to provide 3R functionality.
- All state in the service must be uniquely named by logical identifiers that are assigned when the state is created and never changed. These unique IDs (UIDs) are used to specify the state accessed, modified, or externalized by tracked operations, and are essential in the analysis of external inconsistency.
- Any state affected by an interface verb must be accessible via the interface; this gives the undo system a mechanism for recording the original value of a piece of state, needed to generate the preconditions used to detect inconsistency during replay.
- The interface must offer a complete set of actions on state; it should be possible to generate the inverse of a given verb either with another verb or a sequence of other verbs. This property makes it possible to build compensating actions.
- The service must support relaxed consistency semantics, since external inconsistency is unavoidable with a repair-based undo. Services that demand perfect external consistency and do not allow compensations provide little flexibility for repair during undo.

While these properties may not capture the full set of important service applications today, in many cases only small changes are required to make popular services undo-wrappable. Our target e-mail service is one such example: it already supports relaxed consistency and uses verb-based interfaces (IMAP and SMTP), and with simple extensions (adding an IMAP command to modify message bodies and defining globally-unique message IDs) would fit our undo-wrapping criteria.

5.5. Status

As of this writing, we have just begun a new implementation of the 3R undo architecture described above; we have already built and discarded a partial throw-away implementation of a 3R-undoable e-mail system whose failings inspired many of the design decisions in the architecture presented herein. One of the most significant lessons learned from that early prototype was that it was difficult to have confidence in the implementation because of the mental gymnastics required to anticipate potential external inconsistency and track the state needed to compensate for it. In response, an important goal of our new implementation is to explore the possibility of producing a generic undo wrapper that implements the logging, external consistency management, and 3R functionality in an application-independent manner; it may still be complex, but it will only have to be developed and debugged once. To achieve this goal, we are investigating ways to separate application policy from generic undo mechanism, perhaps by allowing the application to supply specifications of its

verbs and their properties along with callbacks to evaluate preconditions and invoke compensating actions.

6. Conclusions and Future Directions

Traditional approaches to dependability have not eradicated failure and they do not address the problems of operator-induced and operator-compounded failures. To meet the demand for dependable infrastructure systems, we must consider these unavoidable human factors and develop recovery mechanisms that address them. Our system-level undo mechanism does just that: it provides a tool that compensates for the weaknesses of human operators, allows them to erase the effects of their mistakes, and harnesses hindsight to enable retroactive repair, all while preserving the data that users care about.

Although we have taken the first steps toward exploring the issues and challenges associated with implementing system-level undo, there is a great deal more to be done, ranging from a further exploration of the issues raised in Section 4 and their solutions, to extending and formalizing the framework introduced in Section 5, to studying the applicability of the 3R undo model to a broader range of applications, to examining the feasibility of exporting the 3R undo abstraction at a finer granularity to the end-user, to developing a generic implementation of 3R undo as a pluggable framework for services that want undo-based recovery.

There is also great potential for progress in other domains related to undo. One example is problem detection: while psychology shows that humans can quickly self-detect about 70% of problems that they create [15], undo-based recovery would become even more powerful given some mechanism for detecting the remaining 30%. Another example is in techniques to provide virtualized, isolated clones of active systems, allowing the operator to experiment with undo and carry out “what-if” scenarios without affecting the live system. A final area is in benchmarking: new types of *recovery benchmarks* are needed to evaluate the utility of techniques like undo. We are pursuing many of these areas as we develop an implementation of our undo architecture, and would welcome company in further advancing what we see as an essential mechanism for the dependability of tomorrow’s computer systems.

Acknowledgements

The ideas in this paper would not have developed without the input of the members of the UC Berkeley/Stanford ROC research group. Special thanks go to Jim Gray for insightful feedback and inspiration for the framework of Section 5, and to the anonymous reviewers for their comments. This work was supported in part by DARPA under contract DABT63-96-C-0056, the NSF under grant CCR-

0085899 and infrastructure grant EIA-9802069, and the California State MICRO Program.

References

- [1] A. Borg, W. Blau et al. Fault Tolerance Under UNIX. *ACM TOCS*, 7(1):1–24, February 1989.
- [2] A. Brown and D. A. Patterson. To Err is Human. *Proc. 2001 Workshop on Evaluating and Architecting System dependability*, Göteborg, Sweden, July 2001.
- [3] W. K. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. *Proc. 10th ACM Symp. on User Interface Software and Technology*. Banff, Canada, October 1997.
- [4] W. K. Edwards and E. D. Mynatt. Timewarp: Techniques for Autonomous Collaboration. *Proc ACM Conf. on Human Factors in Computing Systems*. Atlanta, GA, March 1997.
- [5] W. K. Edwards, T. Igarashi, et al. A Temporal Model for Multi-Level Undo and Redo. *Proc 13th ACM Symp. on User Interface Software and Technology*. San Diego, CA, November 2000.
- [6] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *CMU TR 96-181*, Carnegie Mellon, 1996.
- [7] P. Enriquez, A. Brown, and D. A. Patterson. Lessons from the PSTN for Dependable Computing. *Proc. 2002 Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, New York, June 2001.
- [8] E. Freeman and D. Gelernter. Lifestreams: A Storage Model for Personal Data. *ACM SIGMOD Bulletin* 25(1):80–86, March 1996.
- [9] E. Gamma, R. Helm, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [10] H. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. *Proc 16th VLDB Conference*, Brisbane, Australia, 1990.
- [11] D. Kurlander and S. Feiner. Editable Graphical Histories. *Proc 1988 IEEE Workshop on Visual Languages*, Pittsburgh, PA, October 1988.
- [12] D. E. Lowell, S. Chandra, and P. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. *Proc. 4th OSDI*. San Diego, CA, October 2000.
- [13] C. Mohan, D. Haderle, et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Systems*, 17(1): 94–162, 1992.
- [14] D. Oppenheimer and D. A. Patterson. Why do Internet services fail, and what can be done about it? *Proc. 10th ACM SIGOPS European Workshop*. Saint-Emilion, France, September 2002.
- [15] J. Reason. *Human Error*. Cambridge University Press, 1990.
- [16] J. Rekimoto. Time-Machine Computing: A Time-Centric Approach for the Information Environment. *Proc 12th ACM Symp. on User Interface Software and Technology*, 1999.
- [17] Roxio, Inc. GoBack3. <http://www.roxio.com/en/products/goback/index.jhtml>.

Session 5. Security & Authentication

1. Brittle Systems will Break Not Bend: Can Aspect-Oriented Programming Help?
Yvonne Coady, Gregor Kiczales
page 78
2. The Case for Cyber Foraging
Authors: M. Satyanarayanan, Rejesh Balan, Shafeeq Sinnamohideen, Jason Flinn, Hen-I Yang
page 87
3. Automatic data dependability
Authors: Kimberly Keeton and John Wilkes
page 93

Brittle Systems will Break – Not Bend: Can Aspect-Oriented Programming Help?

Yvonne Coady, Gregor Kiczales, Joon Suan Ong, Andrew Warfield and Michael Feeley
University of British Columbia

Abstract

As OS code moves to new settings, it must be continually reshaped. Kernel code however, is notoriously brittle – a small, seemingly localized change can break disparate parts of the system simultaneously. The problem is that the implementation of some system concerns are not modular because they naturally crosscut the system structure.

Aspect-oriented programming proposes new mechanisms to enable the modular implementation of crosscutting concerns. This paper evaluates aspect-oriented programming in the context of two crosscutting concerns in a FreeBSD 4.4 kernel – page daemon activation and disk quotas. The ways in which aspects allowed us to make these implementations modular, the impact they have on comprehensibility and configurability, and the costs associated with supporting a prototype of an aspect-oriented runtime environment are presented.

1. Introduction

Simple system rules such as avoiding deadlock by ensuring functions that block are not called while interrupts are disabled, are difficult to verify when inspecting kernel source code [5]. Not surprisingly, the coordination of more complex concerns, such as paging and quotas, are even more challenging to reason about and dependably manipulate. Successfully modifying the implementation of such elements requires both manual inspection of the multiple places in the system where they have impact, and derivation of otherwise implicit semantic information to understand the structural relationships involved. A key part of the problem is that their implementation is not modular.

The goal of aspect-oriented programming (AOP) [11] is to better modularize crosscutting concerns. Ideally, when modularized as aspects, the structure of crosscutting concerns becomes explicit

and hence more comprehensible and configurable. AOP thus has the potential to make OS code less brittle and more amenable to change.

This paper provides an early assessment of our application of AOP to kernel code. After a brief introduction to AOP, the semantics of specific linguistic mechanisms are examined within the context of two practical examples, page daemon activation and disk quotas. An analysis of these implementations then highlights the specific ways in which these aspects provide leverage for reasoning about and configuring crosscutting implementation more comprehensively. Next, the implementation of AspectC runtime support is described, and microbenchmarks of our prototype are provided. Finally, this work is put in context with other research aimed at improving system structure.

1.1. AOP background

The AOP community has proposed linguistic mechanisms intended to allow implementation of crosscutting concerns as first class modules called *aspects*. To enable a range of experiments for operating systems written in C, we are developing AspectC [3]. Conceptually and in syntax, AspectC is a simple subset of AspectJ [10]. Aspect code, known as *advice*, interacts with other code at function call boundaries and can run **before**, **after** or **around** the call to, or execution of the function. The central elements of the language are a means for designating particular function calls, for accessing parameters of those calls, and for declaring advice on those calls. Capturing dynamic control flow context is done using the `cflow` construct, which enables advice to determine the calling context of a function's execution, and access arguments to functions higher up in the call path.

1.2. AOP mechanisms

The following portion of an example aspect, called `profile_low_and_high_level_params`, provides a brief

syntactic and functional introduction to mechanisms advice use to specify when they run, and what values they access.

```
aspect profile_low_and_high_level_params {
    before(int x, int y):
        execution(void low_level(x))
        && cflow(execution(void high_level(y, char)))
    {
        printf("low_level runs with %d when\n",
              high_level runs with %d",
              x, y);
    }
}
```

This aspect defines one advice. Looking at the first line of the advice in detail, this line specifies that the advice executes **before** certain points in the execution of the system, and that it has two integer parameters, `x` and `y`.

The second and third lines – syntactically between the `:` and the `{` – define the execution points to be the intersection of all executions of the `low_level` function, and all functions that execute in the dynamic context of an execution of `high_level`. In other words, executions of `low_level` dynamically within executions of `high_level`. The second line also says that the value for `x` comes from the first argument to `low_level`, and the value for `y` comes from the first argument to `high_level`. The second argument to `high_level` is ignored.

The body of the advice is regular C code. The effect of this advice is print a message and values for `x` and `y` before all executions of `low_level` that occur dynamically within an execution of `high_level`.

2. Examples of Crosscutting Concerns

Examples of crosscutting concerns in kernel code include page daemon activation, disk quotas, prefetching, scheduler activation, checksums, and profiling. This section starts by providing detailed information about the original implementation of page daemon activation in FreeBSD 4.4. This example is then used to show how we have refactored this code as an aspect and structured the implementation in one modular unit. Similarly, the original implementation of disk quotas and a corresponding aspect-oriented implementation of a portion of the original code are shown. The remaining examples are then surveyed, and put in context with related work.

2.1. Page daemon wakeup

In FreeBSD 4.4, the page daemon is responsible for freeing space in the virtual memory system. The page

<i>File</i> [<i>x(frequency)</i>]	<i>Functions</i>
vm/vm_page.c [x4]	vm_page_unqueue vm_page_alloc
vm/vm_fault.c [x1]	vm_fault_additional_pages
kern/vfs_bio.c [x1]	allocbuf

Table 1. Scattering of calls to pagedaemon_wakeup across virtual memory and buffer cache code.

daemon imposes overhead for determining which pages will be replaced and for writing them back to disk if necessary. As a result, timing is an important factor when waking the page daemon – we want to do it only when the number of available pages has fallen below some threshold. The function that wakes the page daemon, `pagedaemon_wakeup`, is invoked from 6 places in the kernel code: 4 calls from VM page operations, 1 call from page fault handling code, and 1 call from buffer allocation. The specific files and functions involved are shown in Table 1.

2.1.1. Page daemon wakeup aspect

We have reimplemented this page daemon wakeup functionality in one aspect. Figure 1 shows the page daemon aspect in its entirety, preceded by a few small helper functions. The code uses named pointcuts to clearly identify specific points in kernel execution when paging may be needed. These four pointcuts, associated with unqueuing pages, allocating pages, faulting pages, and allocating buffers respectively, are shown in the top half of the aspect. Each of the four advice declarations in the bottom half of the aspect uses one of these named pointcuts to say what page daemon wakeup test should happen at each point.

The first pointcut, `unqueuing_available_pages` names points in the execution when `vm_page_unqueue` executes in the control flow of any one of the four functions listed, and makes the `vm_page_t` parameter to whichever of those four functions the execution is within available to advice that uses this pointcut. The first advice executes **around** these points, using the AspectC keyword `proceed` to allow the originally intended function, `vm_page_unqueue`, to execute.

Localizing this implementation in this way allows us to see which global counters are used and when. As highlighted in Figure 1, `cache_min` is not used when faulting, and `free_reserved` is not used when allocating buffers. In the original implementation, establish-


```

/* helper functions */
int pages_available() { return cnt.v_free_count + cnt.v_cache_count; }
int vm_page_threshold() { return cnt.v_free_reserved + cnt.v_cache_min; }
int vfs_page_threshold() { return cnt.v_free_min + cnt.v_cache_min; }

aspect pageout_daemon_wakeup {

  /* when we are unqueuing */
  pointcut unqueuing_available_pages(vm_page_t m):
  execution(void vm_page_unqueue(m))
  && cflow(execution(void vm_page_activate(vm_page_t))
  || execution(void vm_page_wire(vm_page_t))
  || execution(void vm_page_unmanage(vm_page_t))
  || execution(void _vm_page_deactivate(vm_page_t, int)));

  /* when we are allocating new pages */
  pointcut allocating_pages(vm_object_t object, vm_pindex_t pindex, int page_req):
  execution(vm_page_t vm_page_alloc(object, pindex, page_req));

  /* when we are faulting in pages */
  pointcut faulting_pages(int rbehind, int rahead):
  execution(boolean_t vm_page_has_page(vm_object_t, vm_pindex_t, int*, int*))
  && cflow(execution(int vm_fault_additional_pages(vm_page_t, rbehind, rahead,
  vm_page_t*, int*)));

  /* when we are allocating buffer space */
  pointcut allocating_buffers(vm_page_t m, int also_m_busy, const char* msg):
  execution(int vm_page_sleep_busy(m, also_m_busy, msg))
  && cflow(execution(int allocbuf(struct buf*, int)));

  /* below threshold for VM when unqueuing */
  around(vm_page_t m):
  unqueuing_available_pages(m)
  {
    int queue = m->queue;
    proceed(m);
    if (((queue - m->pc) == PQ_CACHE) && (pages_available() < vm_page_threshold()))
      pagedaemon_wakeup();
  }

  /* page alloc fails, or below threshold for VM when allocating */
  around(vm_object_t object, vm_pindex_t pindex, int page_req):
  allocating_pages(object, pindex, page_req)
  {
    vm_page_t allocd_page = proceed(object, pindex, page_req);
    if (allocd_page == NULL)
      pagedaemon_wakeup();
    else
      if (pages_available() < vm_page_threshold())
        pagedaemon_wakeup();
    return allocd_page;
  }

  /* prefetching past modified threshold for VM */
  after(int rbehind, int rahead):
  faulting_pages(rbehind, rahead)
  {
    if ((rahead + rbehind) > (pages_available() - cnt.v_free_reserved))
      pagedaemon_wakeup();
  }

  /* buffer allocating when below threshold for VFS */
  around(vm_page_t m, int also_m_busy, const char* msg):
  allocating_buffers(m, also_m_busy, msg)
  {
    int had_to_sleep = proceed(m, also_m_busy, msg);
    if (!had_to_sleep && ((m->queue - m->pc) == PQ_CACHE)
    && (pages_available() < vfs_page_threshold()))
      pagedaemon_wakeup();
    return had_to_sleep;
  }
}

```

Named pointcuts identify points in the execution of the kernel — paging maybe be needed at any one of these four pointcuts.

Advice declarations make a relationship between advice code and when it runs.

This advice executes around points when we are unqueuing_available_pages(). It has access to vm_page_t m.

Page unqueuing and allocating both use vm_page_threshold() (shown at top of page).

Page fault handling uses a modified threshold, without cnt.v_cache_min

Allocating buffers uses yet another threshold, vfs_threshold() (shown at top of page)

Figure 1. The page daemon wakeup aspect captures the points in the system where the page daemon may be activated if the system is running low on free pages and the possible activation of the daemon at each point.

ing these threshold conditions requires visiting 3 files.

2.1.2. Impact of Page Daemon Aspect

This example shows how an aspect can structure the implementation of page daemon activation within one modular unit. The entire invocation behaviour of the page daemon is captured in this single page of source code. The impact on the paging code is that page daemon activation is no longer scattered in the functions listed in Table 1.

2.2. Quota

Disk quotas are an optional feature of FreeBSD 4.4, configured through a combination of settings in both a kernel configuration file and on a per-file system basis. Through a collection of 37 `#ifdef QUOTA` preprocessor directives in UFS, FFS and EXT2, and 9 `#if QUOTA` directives in EXT2, calculating and maintaining disk quotas is scattered and tangled within 22 functions from 10 files in these file systems, as shown in Table 2. As indicated in the Table, there is overlap between FFS and EXT2 with respect to quota.

Implementing quota with these 46 preprocessor directives supports efficient, coarse grained configurability – we can turn off quota functionality and know it is not part of the binary. Unless we are working directly with quota or code it affects, we can treat this code separately, as it is not part of the core functionality of the file system. Preprocessor directives, however, make it difficult to reason comprehensively about quota, and understand the structural relationships that hold. They also obscure reading of the code quota is scattered in.

2.2.1. Quota aspect

The aspect-oriented implementation of quota localizes the code in a single module. Because aspects can be unplugged from the system by excluding them in the Makefile, the aspect-oriented implementation maintains the same unpluggability as the original preprocessor based implementation.

Figure 2 shows the code for the VFS portion of the aspect-oriented implementation of quota that uses shared advice for FFS and EXT2 – the same quota advice is attached to corresponding functions from the two file systems. As with the daemon activation aspect, the relevant points in the execution of the program are first identified as named pointcuts. The last of these, `vget`, identifies all calls to `ufs_ihashins` within the `cflow` of the execution of either `ffs_vget`

or `ext2_vget`. In this example, the 7 `#ifdefs` in this portion of Table 2 are replaced with 3 advice shown in Figure 2.

2.2.2. Impact of Quota Aspect

Looking at the pointcut declarations, we can see which core file system functions and values are involved, along with their similarities and differences with respect to quota. Looking at the bodies of advice, we see essentially what had been bracketed by preprocessor directives in the original base code. Relative to the preprocessor based implementation, unpluggability has not been compromised. The impact on the rest of the file system code is that the preprocessor directives and associated quota functionality are no longer tangled in the file system functions.

2.3. Advantages of these new perspectives

The intent of our aspect-oriented refactoring is to better separate page daemon activation and disk quotas from the code they crosscut. Aspect-oriented programming naturally involves tool support, similar to that of object-oriented class browsers, that supports easy navigation between aspects and the code they advise. AspectC does not yet support these tools, however, extensions to Emacs, JBuilder, NetBeans and Eclipse are available for AspectJ.

An aspect localizes both the operations of a crosscutting concern, and the declaration of points in the system when the operations happen. The page daemon and quota examples considered here demonstrate how AspectC localizes crosscutting implementation in a way that makes structural information associated with crosscutting explicit. These aspects provide new perspectives that help with page daemon wakeup and file system quota in slightly different ways, respectively.

Knowing when the page daemon may be made runnable makes it easier to reason about activation system-wide. This perspective can be used to more easily ensure a consistent and minimal set of activation points across subsystems.

In particular, seeing the thresholds used to determine daemon activation and the contexts in which they are applied together makes it easier to reason about subtle relationships that exist, such as: (1) page fault handling has the only threshold check that does not use `cache_min`, the minimum number of pages desired on the cache queue, and (2) while VM uses the number of `free_reserved` pages, the number of pages reserved for dealing with deadlock, VFS uses the more conservative value of `free_min`, the minimum number of pages

Component [<i>x(frequency)</i>]	File System		
	UFS	FFS	EXT2
VNode [x21]	ufs_access[x2] ufs_chown[x3] ufs_mkdir[x4] ufs_makeinode[x4]		ext2_mkdir[x4] ext2_makeinode[x4]
VFS [x9]	ufs_quotactl[x1] ufs_init[x1]	ffs_flushfiles[x1] ffs_sync[x1] ffs_vget[x1]	ext2_flushfiles[x2] ext2_sync[x1] ext2_vget[x1]
Inode [x8]	ufs_inactive[x1] ufs_reclaim[x2]	ffs_truncate[x2]	ext2_truncate[x3]
Alloc [x8]		ffs_alloc[x3] ffs_balloc[x1] ffs_reallocg[x1]	ext2_alloc[x3]

overlap

Table 2. Scattering of 46 #ifdef/#if QUOTA across UFS, FFS, and EXT2. The overlap shown in the table refers to identical quota code in both FFS and EXT2.

```

aspect disk_quota {

    pointcut flush(register struct mount *mp, int flags, struct proc *p):
        execution(int ffs_flushfiles(mp, flags, p))
        || execution(int ext2_flushfiles(mp, flags, p));

    pointcut sync(struct mount *mp):
        execution(int ffs_sync(mp, int, struct ucred*, struct proc*))
        || execution(int ext2_sync(mp, int, struct ucred*, struct proc*));

    pointcut vget(struct inode *ip):
        execution(void ufs_ihashins(ip))
        && cflow(execution(int ffs_vget(struct mount*, ino_t, struct vnode**)))
        || (execution(int ext2_vget(struct mount*, ino_t, struct vnode**)));

    ...

    around(register struct mount *mp, int flags, struct proc *p):
        flush(mp, flags, p)
    {
        register struct ufsmount *ump;
        ump = VFSTOUFS(mp);
        if (mp->mnt_flag & MNT_QUOTA) {
            int i;
            int error = vflush(mp, NULLVP, SKIPSYSTEM|flags);
            if (error)
                return (error);
            for (i = 0; i < MAXQUOTAS; i++) {
                if (ump->um_quotas[i] == NULLVP)
                    continue;
                quotaoff(p, mp, i);
            }
        }
        return proceed(mp, flags, p);
    }

    after(struct mount *mp):
        sync(mp)
    {
        qsync(mp);
    }

    before(struct inode *ip):
        vget(ip)
    {
        int i;
        for (i = 0; i < MAXQUOTAS; i++)
            ip->i_dquot[i] = NODQUOT;
    }
    ...
}

```

These pointcuts correspond to file system operations in both FFS and EXT2, on which disk quota advice applies.

Figure 2. Shared advice in the VFS portion of a quota aspect: (a) around prevents ffs/ext2_flushfiles from executing if vflush returns an error; (b) after attaches qsync to ffs/ext2_sync; (c) before attaches quota operations to all executions of ufs_ihashins in the control flow of ffs/ext2_vget.

desired to be kept free.

Though the rationale behind the different thresholds is not immediately apparent to a non-expert and is not documented in the original implementation, it is clear that thresholds are context sensitive. Coalescing them into one module brings these differences to light. Because they appear side by side, it should be easier for the original implementor to document the rationale.

With respect to disk quotas, looking at the point-cut declarations in Figure 1 is like looking at a more detailed version of the structural relationships outlined in Table 2. We can see which core file system functions and values are involved, along with their similarities and differences with respect to quota. We can now reason about and configure quota across these file systems, sharing its implementation where appropriate.

This perspective can be used to eliminate redundancy and more easily ensure the consistent application of quota operations across file systems. In particular, half the code indicated by the overlap in Table 2 can be eliminated because it can now be shared between file systems.

2.4. Other crosscutting concerns

In addition to page daemon activation and disk quotas, we are exploring the aspect-oriented implementation of elements of prefetching, scheduling, networking and profiling.

Prefetching involves coordination between high level allocation of pages in VM, and subsequent possible low level deallocation in file systems. Tracing the page-fault path in the FreeBSD v3.3 implementation requires traversing 5 files, 2 levels of function tables, and 4 changes in variable names. Our preliminary aspect-oriented refactoring of normal and sequential prefetching as path-specific customizations is reported in [6].

Scheduling code spans interrupt handlers, device drivers, and process synchronization. One of the challenges in the development of Bossa, a domain specific language for schedulers, was to precisely identify all the scheduling points, or circumstances under which the scheduler is activated throughout the OS [4]. Extending the scheduler to respond to Bossa-defined scheduling events requires access to the context of the scheduler invocation. To get an idea of how extensive the challenge is to track this context, Table 3 shows the functions that call the scheduler with `mi_switch` in FreeBSD 4.4, along with the number of places where those callers are called.

Networking involves some concerns that run the length of the protocol stacks of communicating pro-

Callers of <code>mi_switch</code>	Number of Calls to Callers
<code>tsleep</code>	483
<code>await</code>	19
<code>exit</code>	95
<code>issignal</code>	95
<code>uio_yield</code>	4
<code>usrret</code>	9

Table 3. Control Flow to `mi_switch` in FreeBSD 4.4

cesses. Optimizing or customizing a protocol often requires introducing integrated layer processing and/or passing additional parameters to new functionality introduced at each layer. Something as simple as suppressing checksums to improve performance of consenting processes not concerned with data integrity requires symmetrical changes to the sending and receiving stacks. One of the contributions of both the *x*-kernel [8], a framework for implementing network protocols, and later Plexus [7], an extensible protocol architecture for application-specific networking, was the use of protocol graphs for representing standard protocols, with augmentation for modified functionality. These systems used these graphs to represent new protocols in terms of a cohesive set of related modifications to the standard.

Profiling inherently involves action at a variety of points in the system. Whether it be for tracing execution, verifying system rules, or as a basis for building more sophisticated gray-box information and control layers [2], the ability to build a comprehensive profile is a prerequisite for dependable systems. Preprocessor directives are commonly used to introduce unplugable profiling code in the kernel. The `/dev/usb` directory in FreeBSD 4.4 contains approximately 50 such `#ifdef DIAGNOSTIC` statements scattered throughout roughly 10,000 lines of code. System-wide, there are 314 `#ifdef DIAGNOSTIC` directives.

3. AspectC runtime

Like AspectJ, most AspectC constructs are resolved at compile time. They introduce no more overhead than a call to an inlineable function containing the advice body. (Though the pre-processor could inline these directly, it currently does not, in order to help make the pre-processor output more readable.)

But `cflow` is a dynamic construct and hence has runtime overhead associated with it. We follow the AspectJ implementation model for `cflow`, in which the overhead is distributed across executions of functions

Granularity	Cflow Function	Overhead (nanoseconds)
per-process	cflow_add_pid_entry	777
	cflow_del_pid_entry	141
per-call	cflow_push	79
	cflow_pop	80
	cflow_test	86
	cflow_get	73

Table 4. Microbenchmarks for core cflow overhead.

that are *cflow-tested*, and dispatch to advice involving a cflow test.

In the example from Section 1.2, a `cflow_push` and `cflow_pop` are effectively added to the code for `high_level`. A `cflow_test` is effectively added to `low_level`, as part of testing whether the advice should run. If the advice does run, `cflow_get` is called to access the parameters. These push/pop/test/get operations would all use a process-local stack specifically dedicated to `high_level`.

Our current implementation of the push/pop/test/get runtime routines is trivially naive. An open hash table tracks this information on a per-process basis. A pool of entries, sufficiently large to track the maximum number of processes in the system, is statically allocated at boot time. Each entry tracks the necessary cflow information for a single process, uniquely identified by the process identifier (PID).

Table 4 provides microbenchmarks for our prototype AspectC runtime. These benchmarks were taken on a 700MHz Pentium-III processor. The first two rows in the Table show the costs of adding and deleting hash table entries during process initialization and tear-down. The next four rows show the per-call costs of the other push/pop/test/get routines.

4. Future work and open issues

In order to more rigorously assess the impact aspects have on kernel code, issues of scalability, configurability, extensibility, evolvability and performance require in depth cost/benefit analysis. Improving modularity of OS kernel code will not be meaningful if aspects substantially adversely impact performance. Specifically, we need to know the costs associated with sophisticated compositions of aspects in terms relative to a tangled implementation. In kernel code, we are often faced with a fine granularity of tangling of multiple concerns, making refactoring challenging. For example, IP security (IPSEC) and IPv6 functionality (INET6), are

configured with compiler directives and implemented using a shared `if` statement and `goto` labels as shown below:

```
#ifdef IPSEC
#ifdef INET6
if (isipv6) {
    if (inp != NULL &&
        ipsec6_in_reject_so(m,
            inp->inp_socket)) {
        ipsec6stat.in_polvio++;
        goto drop;
    }
} else
#endif /* INET6 */
if (inp != NULL && ipsec4_in_reject_so(m,
    inp->inp_socket)) {
    ipsec4stat.in_polvio++;
    goto drop;
}
#endif /* IPSEC */
```

Though AspectC is modeled after AspectJ, there are several important differences that must be addressed. This includes C-specific issues, such as code-bloat associated with the C preprocessor. Working within the kernel may also demand that we explore different kinds of runtime support than is required for user-level AOP.

5. Related work

Structuring kernel code to make it more amenable to change common theme in many research projects. Customization has been a leading motivating factor. Support for application-specific customization of services range from operating systems that target specific policy, such as paging in Mach [13], to those that have taken a more comprehensive approach, such as the use of reflection in Apertos [21]. Approaches that structure client participation in OS policy include scheduler activations [1], active networking [18], policy servers in user space [20, 9, 12], and application-specific extensions [15, 7, 17, 19]. Approaches aimed at improving structure in general include the use of frameworks for end-to-end optimization [16], domain specific languages [14, 4], and gray-box techniques [2].

Our work is particular in its focus on the modular implementation of existing crosscutting concerns that map to key decisions in kernel design.

6. Conclusions

One of the reasons kernel code is brittle is that some key system concerns naturally crosscut others. Crosscutting concerns are not modular when implemented using traditional techniques – their implementation is scattered and tangled throughout other modular units of the system.

AOP is poised to help. It offers mechanisms that allow us to explicitly structure crosscutting concerns

as new, first class modules called aspects.

In this paper, a subset of AOP mechanisms are applied in the context of two examples, page daemon activation and disk quotas. The benefits of implementing these particular concerns as aspects are improved comprehensibility and configurability. By further studying the ways in which aspects can be used to improve the modularity of key design decisions in an existing kernel, we hope to promote dependability in future systems.

References

- [1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1), February 1992.
- [2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *18th ACM Symposium on Operating System Principles (SOSP)*, 2001.
- [3] AspectC. www.cs.ubc.ca/labs/spl/aspects/aspectc.html.
- [4] L. P. Barreto and G. Muller. Bossa: a language-based approach for the design of real time schedulers. In *Proceedings of the 23rd IEEE Real-Time Systems*, 2002.
- [5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *18th ACM Symposium on Operating System Principles (SOSP)*, 2001.
- [6] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, 2001.
- [7] M. E. Fiuczynski and B. N. Bershad. An extensible protocol architecture for application-specific networking. In *Winter Usenix Conference*, 1996.
- [8] N. Hutchinson and L. Peterson. The x-kernel: An architecture for implementing network protocols. In *IEEE Transactions on Software Engineering*, volume 17(1), 1991.
- [9] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.
- [10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. AspectJ home page. <http://www.aspectj.org>.
- [11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.
- [12] C. Maeda. Flexible system software through service decomposition. In *OOPSLA*, August 1994.
- [13] D. McNamee and K. Armstrong. Extending the Mach external pager interface to allow user level page replacement policies. In *Technical Report UWCE 90-09-05, University of Washington*, September 1990.
- [14] G. Muller, C. Consel, R. Marlet, L. P. Barreto, F. Merillon, and L. Reveillere. Toward robust oses for appliances: A new approach based on domain-specific languages. In *European Workshop on Operating Systems*, 2000.
- [15] P. Pardyak and B. Bershad. Dynamic binding for an extensible system. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.
- [16] A. Sane, A. Singhai, and R. Campbell. Framework design for end-to-end optimization. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.
- [17] C. Small and M. Seltzer. A comparison of OS extension technologies. In *Proceedings of the USENIX Conference*, 1996.
- [18] D. Tennenhouse and D. Wetherall. Towards an active network architecture. *ACM Computer Communications Review*, 26(2):5–18, April 1996.
- [19] A. C. Veitch and N. C. Hutchinson. Kea - a dynamically extensible and configurable operating system kernel. In *Proceedings of the 1996 Third International Conference on Configurable Distributed Systems (IC-CDS)*, 1996.
- [20] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. In *Communications of the ACM*, volume 17(6), 1974.
- [21] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1992.

The Case for Cyber Foraging

Rajesh Balan^{†,*}, Jason Flinn[‡], M. Satyanarayanan^{†,‡}, Shafeeq Sinnamohideen^{†,‡} and Hen-I Yang[†]
[†]Carnegie Mellon University and [‡]Intel Research Pittsburgh
 rajesh@cs.cmu.edu

Abstract

In this paper, we propose cyber foraging: a mechanism to augment the computational and storage capabilities of mobile devices. Cyber foraging uses opportunistically discovered servers in the environment to improve the performance of interactive applications and distributed file systems on mobile clients. We show how the performance of distributed file systems can be improved by staging data at these servers even though the servers are not trusted. We also show how the performance of interactive applications can be improved via remote execution. Finally, we present VERSUDS: a virtual interface to heterogeneous service discovery protocols that can be used to discover these servers.

1. Introduction

The designers of mobile computing devices face a never-ending dilemma. On the one hand, size and weight are dominant factors. The need to make mobile devices smaller, lighter and long-running compromises their computing capabilities. On the other hand, user appetites are whetted by their desktop experiences. Meeting their ever-growing expectations requires computing and data storage ability beyond that of a tiny mobile computer with a small battery. How do we reconcile these contradictory demands?

We conjecture that *cyber foraging*, construed as “living off the land”, may be an effective way to solve this dilemma [8]. The idea is to dynamically augment the computing resources of a wireless mobile computer by exploiting nearby compute and data staging servers. Such infrastructure may be discovered and used opportunistically at different locations in the course of a user’s movements. When no such infrastructure is available, the mobile computer offers a degraded but acceptable user experience. Using higher-level knowledge, it may also identify nearby locations that might offer a better user experience.

*School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA

Who will provide the infrastructure for cyber foraging? Desktop computers at discount stores already sell for a few hundred dollars, with prices continuing to drop. In the foreseeable future, we envision public spaces such as airport lounges and coffee shops being equipped with compute and data staging servers for the benefit of customers, much as comfortable chairs and table lamps are provided today. These will be connected to the wired Internet through high-bandwidth networks.

When hardware in the wired infrastructure plays this role, we call it a *surrogate* of the mobile computer it is temporarily assisting. Two important attributes of surrogates are that they are *untrusted* and *unmanaged*. These are key assumptions because they reduce the total cost of ownership of surrogates and hence encourage their widespread deployment. It is the responsibility of mobile clients to establish adequate trust in the surrogates they choose to use.

In this paper we explore the concept of cyber foraging and discuss the research challenges posed. We also describe the status of our research in this area. Specifically, we illustrate how the use of surrogates can help in two distinct situations. First, we show how data staging can reduce cache miss service times in mobile file access. Second, we show how remote execution enables compute-intensive applications like language translation and augmented reality to run on mobile hardware.

2. Usage Scenario and Research Challenges

We envision a typical scenario as follows. When a mobile computer enters a neighborhood, it first detects the presence of potential surrogates and negotiates their use. Communication with a surrogate is via short-range wireless technology. When an intensive computation accessing a large volume of data has to be performed, the mobile computer ships the computation to the surrogate; the latter may cache data from the Internet on its local disk in performing the computation. Alternatively, the surrogate may have staged data ahead of time in anticipation of the user’s arrival in the neighborhood. In that case, the surrogate may perform computations on behalf of the mobile computer or

merely service its cache misses with low latency by avoiding Internet delays. When the mobile computer leaves the neighborhood, its surrogate bindings are broken, and any data staged on its behalf are discarded.

This usage scenario exposes many important research questions. Here are some examples:

- What is the system support needed to make surrogate use seamless and minimally intrusive for a user? What parts of this support are best provided by the mobile client, and what by the infrastructure?
- How much advance notice does a surrogate typically need to act as an effective staging server? Is this on the order of seconds, minutes or tens of minutes? What implications does this requirement have for the other components of a pervasive computing system?
- How does one establish an appropriate level of trust in a surrogate? What are useful levels of trust in practice? How applicable and useful is the concept of caching trust [7]? Can one amortize the cost of establishing trust across many surrogates in a neighborhood?
- How is load balancing on surrogates done? Is surrogate allocation to be done based on an admission control or best-effort approach? How relevant is previous work on load balancing on networks of workstations [2]?
- What are the implications for scalability? How dense does the fixed infrastructure have to be to avoid overloads during periods of peak demand?
- How does one discover the presence of surrogates? Of the many proposed service discovery mechanisms such as JINI [10], UPnP [6], and Bluetooth proximity detection [1], which is best suited for this purpose? Can one build a discovery mechanism that subsumes all of them for greatest flexibility?
- How do we deal with unmanaged surrogates? If they are used for remote execution, how can a mobile client ensure that the remote executing environment is correctly configured? What role, if any, can virtual machines such as VMware [4] play in establishing suitable execution environments? Can virtual machines also help with establishing trust?

This is obviously a very broad and ambitious research agenda, and our current work only addresses a subset of these questions. We summarize the status of our research in the rest of the paper. Section 3 describes how we are using data staging on surrogates to reduce the latency of servicing cache misses. Section 4 discusses our use of surrogates for remote execution. Note that trust is an issue that we

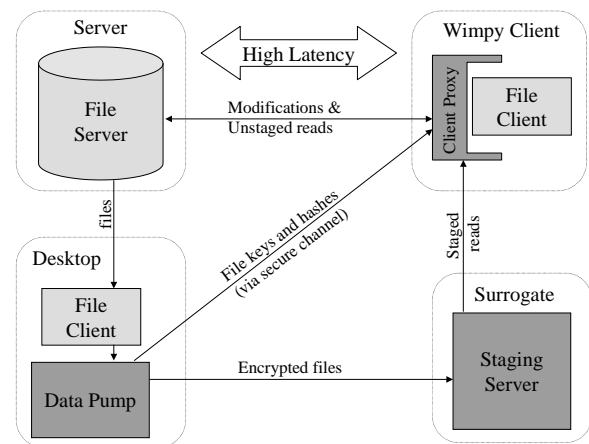


Figure 1. Data Staging Architecture

have addressed in the context of data staging, but not yet in the context of remote execution. Section 5 describes our platform-independent approach to surrogate discovery.

3. Using Surrogates for Data Staging

3.1. Background

Can an untrusted computer facilitate secure mobile data access? Surprisingly, the answer is “yes.” Data staging enables untrusted, unmanaged computers to be used to improve the performance of cache miss handling in an Internet-wide distributed file system. The untrusted computer, called a *surrogate*, plays the role of a second-level file cache for a mobile client. By proactively staging data on the surrogate, cache misses from a nearby mobile client can be serviced at low latency (typically one wireless hop) instead of full Internet latency.

A coast-to-coast ping in the United States typically takes about 60 ms., well above the speed-of-light bound of about 32 ms. The growing use of firewalls adds per-packet processing, and hence delay, to each packet. The impact of latency on distributed file systems is easily seen in interactive file-intensive applications such as mail readers, directory browsers, and digital photo albums. In such applications, a flurry of serial cache misses on relatively small files can result in annoying delays and sluggish behavior. Merely improving bandwidth does not help such interactive applications because they are latency-limited rather than bandwidth-limited.

Although systems such as Coda [9] have shown that hoarding can reduce cache misses and hence mask network

latency, there are limits to its effectiveness. First, size and weight restrictions may result in the flash memory or disk on a mobile computer being too small to hoard all relevant data. Second, some files that were not hoarded may unexpectedly become relevant to the user. Third, some files may be updated by other users and thus result in cache misses when accessed after a period of disconnection. These circumstances all lead to unavoidable cache misses and hence poor interactive performance.

Data staging helps solve this problem by speculatively prefetching distant data to nearby surrogates. In effect, clients borrow the storage capacity of surrogates and use it as a secondary file system cache. Cache misses from the mobile client are serviced by a *staging server* running on the nearby surrogate rather than by the distant file server.

3.2. Current Status

We have built an initial implementation of data staging for Coda. Figure 1 shows how our architecture is split across four computers: the file server (unmodified Coda server), the surrogate, a home desktop machine, and the mobile client. The client and surrogate are located close together and are typically connected by a low-latency wireless connection such as 802.11, Bluetooth, or infrared. The file server is distant, so network communication to and from the file server incurs high latency. A proxy located on the client intercepts and redirects file system traffic. If a request is for data contained on a nearby staging server, the proxy directs the request to the surrogate. Otherwise, it forwards the request to the distant file server.

Surrogates are untrusted—we therefore use end-to-end encryption to ensure privacy and secure hashes to guard against malicious modification of file data. The client proxy initiates the staging of data by contacting a data pump running on the end user's desktop machine. The client proxy specifies a list of files to stage—the data pump reads these files from the distributed file system, encrypts them with per-file keys, generates a secure hash of the data, and sends the encrypted files to the staging server. The data pump also sends the keys and hashes for the staged files to the client proxy using a secure channel. When an application reads data that is staged on the surrogate, the client proxy reads the file from the surrogate, decrypts it, verifies that it has not been modified using the secure hash, and returns the data to the application. Since storage requirements are small (at most 72 bytes per file), it is feasible to hoard keys and checksums even when the data itself is much too large to hoard.

Another important focus of our work is making surrogates as easy to manage as possible. We envision surrogates that are as simple as table lamps; they should require no system administrator or complex maintenance procedures.

We have identified two design principles that improve ease of management. The first principle is to build as much as possible upon widespread commodity software, so as to leverage the improved reliability that comes through the extensive testing provided by a large user community. To this end, we use the Apache Web server as the base system for our surrogates. We have identified the minimum set of additional functionality that must be located on the surrogate, and provide this functionality with CGI scripts. All other functionality is pushed to the client proxy and data pump in order to keep the custom code base on the surrogate as simple and reliable as possible.

The second design principle is to maintain no long-term state on the surrogate. For example, we do not buffer client modifications to file data on surrogates. Since all state is soft, no critical information is lost if the surrogate is disrupted by power failure or a system crash. The only consequence of surrogate failure is that clients receive the same sub-par performance for file accesses that they would have received if the surrogate had not been present in the first place.

3.3. Work in Progress

We have built a prototype of the data staging architecture that uses Coda as the base distributed file system. We are currently developing clients for both x86 (laptop) platforms, as well as StongArm (handheld) platforms. We have validated the prototype by replaying recorded traces of file system accesses—our results show that data staging can reduce the cumulative delay due to distributed file system accesses by up to 64%.

Our future work will explore the following questions:

- What is the best prediction algorithm for deciding which data to stage on a surrogate?
- What is the right cache management scheme?
- How will data staging benefit other distributed file systems such as NFS?

4. Using Surrogates for Remote Execution

4.1. Background

Remote execution for pervasive computing must reconcile multiple, possibly contradictory, goals. For example, executing a code component on a remote server might reduce client energy usage at the cost of increasing execution time. Also, due to the dynamic nature of the environment, it is not feasible to use static policies to determine how and where to remotely execute applications as the current resource situation may obsolete any statically chosen policy.

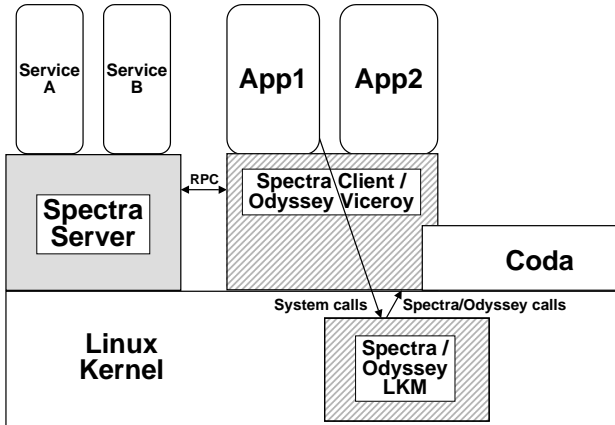


Figure 2. Spectra Architecture

Hence, the decision of how and where to remotely execute a code component must be determined dynamically based on the current resource availability in the environment.

Our work addresses the following aspects of remote execution:

- Monitoring resource availability
- Making dynamic remote execution decisions based on resource availability
- Simplifying the task of modifying applications to use remote execution
- Automatically using extra available resources in an over-provisioned environment to improve application performance

We currently trust surrogates used for remote execution. Developing the mechanisms for establishing this trust remains future work.

4.2. Current Status

We have implemented a system called Spectra [3] that monitors the current resource availability and dynamically determines the best remote execution plan for a given application. To make this decision, Spectra measures the supply and demand for many different resources such as bandwidth, file cache state, CPU, and battery life. Figure 2 illustrates Spectra's architecture.

Spectra targets applications that perform relatively coarse-grained operations of a second or more in duration. These applications include speech recognition, augmented reality, and language translation. Spectra tries to

meet the application goals in the light of available resources. However, application goals can frequently be contradictory. For example, an application might require a high network throughput (which requires excessive use of a power hungry wireless network card) while also requiring low battery usage. Thus, Spectra has to reconcile these contradictory goals when making a remote execution decision.

To make good decisions, Spectra must predict the resource usage of alternative execution plans for an application. It does this through an approach called *self-tuning*, where it records the history of resource usage by an application and uses machine learning techniques to predict future usage.

4.2.1. Chroma

The major drawback of Spectra is that application developers must explicitly modify their applications to use Spectra. This hurts software maintenance and portability. We are currently building a new remote execution system, called Chroma, that subsumes the functionality of Spectra. Chroma addresses the shortcomings of Spectra by separating the adaptive policies of an application from the actual decision making and enforcing of the policy at runtime.

Chroma is based on three observations derived from experience with Spectra:

- Most applications for mobile devices can be created by modifying existing applications rather than writing new applications from scratch.
- The modifications for adaptation typically affect only a small fraction of total application code size. Much of the complexity of implementing adaptation lies in understanding the base code well enough to be confident of the changes to make.
- The changes for adaptation can be factored out cleanly and expressed in a platform-neutral manner.

Based on these observations, Chroma focuses on reducing the effort required to make applications use remote execution. It consists of three parts:

- A lightweight semi-automatic process for customizing the adaptation API used by the application. Such customization is targeted to the specific adaptation needs of each application.
- A tool for automatic generation of code stubs that map the customized API to the specific adaptation features of the underlying mobile computing platform.
- Run-time support for monitoring resource levels and triggering adaptation is factored out of applications into a set of operating system extensions for mobility.

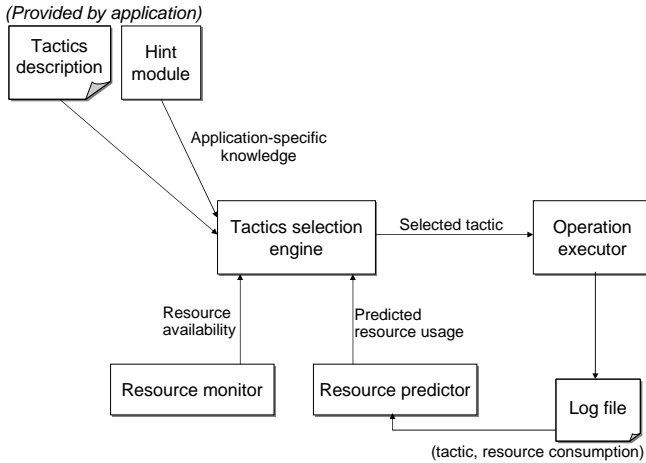


Figure 3. Chroma Run-time Components

4.2.2. Over-Provisioned Environments

Most remote execution systems are designed to operate in environments where resources are either scarce or just adequate. However, with the emergence of smart rooms and the dramatic decrease in the price of computing, environments where resources are over-provisioned are becoming a reality. These extra resources can be used to counter the variability inherent in mobile environments. When using just one remote server to perform an operation, any transient load spikes or network glitches affecting that server can have a dramatic effect on application performance. By using more than one server to perform the same operation, the effect of these load spikes and network glitches is minimized as the fastest result is returned to the application. Thus even if one server is experiencing transient load, other servers may be unloaded and will return a result faster than the loaded server (assuming all the servers are the same). One of the design goals of Chroma is to automatically use these extra resources to improve the performance of applications.

Chroma uses the notion of *tactics* to achieve this goal. Tactics are a concise declarative description of an applications remote execution capabilities and express the useful remote partitions of an application. Using this information, Chroma can automatically execute different parts of the application opportunistically on extra resources to achieve better application performance.

Figure 3 shows the run-time components used to handle tactics. Chroma determines the predicted resource usage of the tactics of the application by querying the resource prediction module. At the same time, Chroma determines the available resources via the resource measurement module.

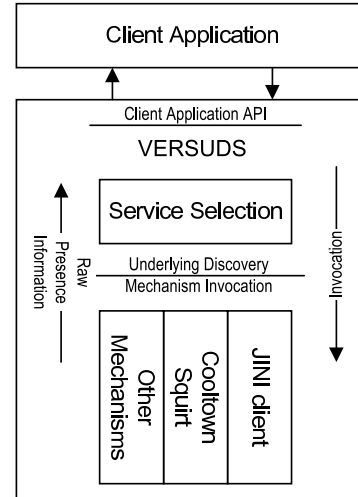


Figure 4. VERSUDS Architecture

Chroma then decides how to remotely execute this application by finding the best match between available resources and the predicted resource usage of each tactic for an operation. Chroma also determines if extra resources are available in the environment. If they are, Chroma will opportunistically use these resources to remotely execute the application. We have shown that using tactics to opportunistically use extra resources in the environment can result in tremendous improvement in application response time.

4.3. Work in Progress

We are currently working on mechanisms to make Chroma even easier to use for application developers. Our use of a customized API coupled with automatic code generation is a major step in that direction. However, more work needs to be done in this area.

We also continuing our research in the use of tactics as a method of exploiting over-provisioned environments. We are investigating resource allocation policies that will ensure a fair use of these extra resources among multiple clients.

Finally, we are looking at existing work on security and integrity of remote servers. We hope to be able to use some of this work in our system.

5. Discovering Surrogates

The previous two sections have detailed our work in using surrogates for staging data and for remote execution. However, before either of these two operations can be performed, it is necessary to first discover the presence of surrogates in the current environment.

We have implemented a versatile surrogate discovery service (VERSUDS) that uses existing heterogeneous service discovery mechanisms to detect the presence of neighbour surrogates. VERSUDS provides a virtual layer that sits on top of these existing service discovery mechanisms. It provides a standardized API to applications and thus isolates applications from having to deal with different service discovery mechanisms as the environment changes. VERSUDS automatically translates application requests to the format of the underlying service discovery mechanism and vice versa. The VERSUDS architecture is shown in Figure 4.

VERSUDS currently provides support for JINI and Cooltown [5]. It also supports application-provided filters that specify which resources the application is interested in discovering. These filters can be used to specify dynamic system resource attributes such as CPU utilization and available memory as well as specific static attributes such as administrative domain and service price. We are currently extending its capability to support more complex application-specific filters and to support other service discovery mechanisms.

6. Conclusion

We have described our initial research in the area of cyber foraging. We have shown that data staging is able to provide improved latency for file access without requiring the staging servers to be trusted. We have described our remote execution system, Chroma, and highlighted the key aspects that make Chroma easy to use for application developers. We have also described how Chroma is able to use extra resources in the environment to improve application performance. Finally, we have described our initial work in developing an environment independent surrogate discovery service.

7. Acknowledgments

This research was supported by the National Science Foundation (NSF) under contracts CCR-9901696 and ANI-0081396, the Defense Advanced Projects Research Agency (DARPA) and the U.S. Navy (USN) under contract N660019928918. Rajesh Balan was additionally supported by a USENIX student research grant. We would also like to thank Hewlett-Packard for donating notebooks to be used as servers and Compaq for donating handhelds to be used as clients. Finally, we would like to thank Dushyanth Narayanan, SoYoung Park, Tadashi Okoshi, Bradley Schmerl and Joao Sousa for their many insightful comments and suggestions related to this work. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, DARPA, USN, HP, USENIX, Compaq, nor the U.S. government.

References

- [1] 3COM, Agere, Ericsson, IBM, Intel, Microsoft, Motorola, Nokia and Toshiba. *Bluetooth Wireless Information Site*, 1999. <http://www.bluetooth.com>.
- [2] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Philadelphia, Pennsylvania, USA, May 1996.
- [3] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.
- [4] L. Grinzo. Getting Virtual with VMware 2.0. *Linux Magazine*, June 2000.
- [5] T. Kindberg, J. Barton, J. Morgan, G. Becker, I. Bedner, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, C. Perring, J. Schettino, B. Serra, and M. Spasojevic. People, places, things: Web presence for the real world. In *Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMSCA 2000)*, Monterey, California, USA, Dec. 2000.
- [6] Microsoft Corporation. *Universal Plug and Play Forum*, June 1999. <http://www.upnp.org>.
- [7] M. Satyanarayanan. Caching Trust Rather Than Content. *Operating System Review*, 34(4), October 2000.
- [8] Satyanarayanan, M. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4), August 2001.
- [9] Satyanarayanan, M. The Evolution of Coda. *ACM Transactions on Computer Systems*, 20(2), May 2002.
- [10] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.

Automating data dependability

Kimberly Keeton and John Wilkes

Storage Systems Department, Hewlett-Packard Laboratories, Palo Alto, CA, USA

{kkeeton,wilkes}@hpl.hp.com

Abstract

If you can't make your data dependable, then you can't make your computing dependable, either. The good news is that the list of data protection techniques is long, and growing. The bad news is that the choices they offer are getting more complicated: how many copies of data to keep? whether to use full or partial redundancy? how often to make snapshots? how to schedule full and incremental backups? what combination of techniques to use? The stakes are getting higher: web access means that services must have 24x7 availability, and users are willing to switch if services are unavailable. Finally, human administrators can (and often do) make mistakes. These factors compel us to simplify and automate data dependability decisions as much as possible.

We are developing a system that will automatically select which data protection techniques to use, and how to apply them, to meet user-specified dependability (i.e., reliability and availability) goals. This paper describes our approach and outlines our initial descriptions for user requirements, failure characteristics and data protection techniques.

1 Motivation

A dependable system is one that just works: it does what you want, when you want it, to meet your needs. More and more, those needs are associated with access to information – without that information, highly dependable endpoints are useless. Getting it wrong is expensive: e-commerce sites, such as Amazon.com and ebay.com, may lose up to \$200,000 per hour of downtime, and financial services may lose as much as \$2.5M to \$6.5M per hour of downtime [11].

The information had better be reliable, too – “losing” data is even worse than failing to provide access to it when it is needed. Many businesses, including financial institutions, pharmaceutical companies, and trading companies, must retain data for multi-year periods to meet legal requirements. As a result, losing data may have far-reaching ramifications.

This paper describes our approach to the data dependability problem. We concentrate on enterprise-scale storage systems, because their information needs affect so many of today's computing services. Today's enterprise storage systems range in size from many terabytes to a few petabytes of storage, with high rates of growth. Storage users demand predictable performance, very high availability, and extreme levels of data reliability. The sheer size of the systems means that administrators want more cost-effective and high-performance solutions than the traditional solution of “copy everything to tape.” Administrators also want techniques to protect against important problems, such as user and software errors. For example, in today's SAN-connected systems, mirroring defends against device failure, but still propagates mistakes or software errors.

Over the last decade, the set of data protection techniques has increased significantly. In addition to traditional tape-based backup, online techniques using high-density disks and incremental snapshots are

becoming attractive, as the price of disk storage capacity plummets [10]. Geographic distribution, replication and partial redundancy through RAID levels or erasure codes have also been enabled by cheaper wide area network performance.

Each technique provides some portion of the protection that is needed; combined, they can cover a much broader range. The key question, then, is how to determine the appropriate combination of different techniques to provide the data protection desired by the user. For example, one popular combination includes local RAID-5, remote mirroring, snapshot and backup to tape. RAID-5 provides protection against disk failures, remote mirroring guards against site failures, snapshots address user errors, and tape backup protects against software errors and provides archival.

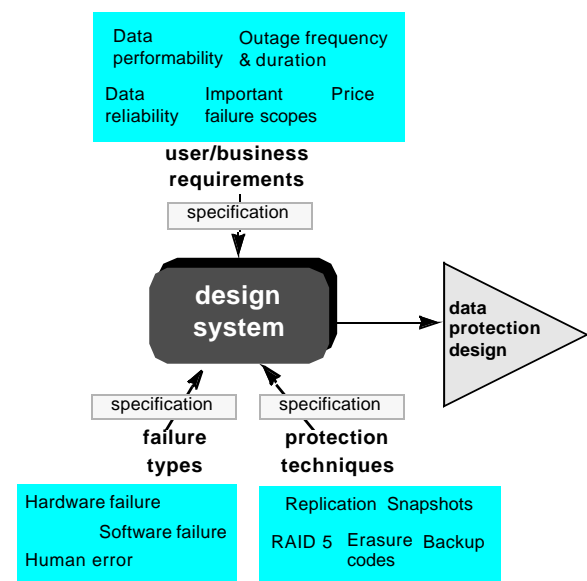


Figure 1: our approach to automating data dependability.

Selectivity in picking techniques matters – assigning different levels of protection to different kinds of information can save money or free up resources for providing more protection to important data. The mixture of techniques needs to change over time as user and business requirements change, as the environment changes and as new techniques become available. Unfortunately, the mix of techniques is too rich for people to reason about well, with the result that they either grossly over-provision, or accidentally omit coverage for events that later destroy their data. The former is merely expensive – storage systems typically comprise about half of the capital cost of current computer systems, with even higher operational costs. The latter can be catastrophic.

2 Our approach

We believe there is a better way: instead of asking humans to perform large-scale, complicated optimization problems, get the storage design system to help:

- Instead of asking people to describe how to implement data protection, have them specify their needs to the storage system, and let the system determine which implementation best meets them.
- Instead of asking people to reason about infrequent failure events, have them specify how available and reliable data must be, and let the system figure out which failures must be handled to meet these goals.
- Instead of expecting people to choose between available techniques and reason about their side-effects and interactions, let the system automatically choose the appropriate techniques, and incorporate new techniques as they become available.
- Instead of expecting people to monitor the system, and adapt its implementation to unexpected changes such as increased failure rates for a component, get the storage system to do it.

Figure 1 illustrates this automatic goal-directed design process. We have successfully applied this methodology to performance-related storage goals [1, 2, 3]. We believe the time is right to extend it to dependability. In particular, we concentrate on the *reliability* (whether the system discards, loses or corrupts data) and *performability* (whether the data can be accessed at a particular performance level at a given time) aspects of dependability [5].

We focus here on reliability and performability for data, rather than for applications. Even though users ultimately care about their end-to-end service needs, data dependability is a prerequisite for application dependability. We choose to start here because the narrower storage interface may provide more straightforward solutions, and these solutions will benefit a wide range of applications. We begin by examining the block-level storage system, and plan ultimately to extend the approach to higher-level data from file systems and databases.

2.1 Automating data dependability

Our approach to automating this problem has five main components (see Figure 1):

1. A description of the user's requirements (a *requirements specification*) for data performability and data reliability.
2. A description of the *failures* to be considered, including their scope and likelihood of occurrence.
3. A description of the *data protection techniques* available, including what failures they can tolerate, and how they recover from these failures.
4. A *design system*: a set of tools, including ones that select which techniques to use to satisfy the requirements.
5. The output is a *data protection design* specifying which data protection techniques should be deployed against which pieces of data and parameter settings (e.g., frequencies, retention times, etc.)

We note that the user is involved in only the first component, specifying the data protection requirements. Users may optionally designate that certain failures are important (e.g., electric utility unreliability may be problematic at a site), or conversely, unimportant (e.g., hurricanes in California are infrequent). They may also optionally express preferences for certain data protection techniques over others (e.g., from manufacturer X instead of Y). The underlying storage design system is responsible for enumerating and describing failures and the behaviors of the data protection techniques.

To automate the process of data dependability, our specifications for user requirements, failures and data protection techniques are made in a quantitative, declarative fashion [16]. Although the storage design system can work with only partial specifications for any of the components, more detailed information will lead to higher quality data protection designs.

2.2 The storage design system

The storage design system is a suite of tools that chooses (or, initially, assists with choosing) data protection techniques that meet the specified goals. We view the problem as an optimization one, by analogy to our earlier work on storage system design for performance [1, 3]. We envision a solution including the following tools:

- *user interrogation tool*: a graphical user interface (GUI) that asks users what they want, and then maps their intuitively specified goals into a quantitative specification. This tool removes the need for users to supply reams of numbers.
- *design checker*: this tool applies well-known modeling techniques [9] to predict whether a data protection design will satisfy a set of user goals.
- *design comparator*: using the design checker, this tool determines which of two designs more effectively meets the user's goals.

- *design tool*: this tool uses optimization techniques to automatically find the best data protection design to satisfy the user’s goals, employing the design comparator to compare alternative designs [1, 3].

The design tool produces a data protection design, which records the design decisions, including what techniques should be applied to each data object and how the technique configuration parameters should be set. Given such a design, several additional management tools are desirable:

- *configuration tool*: this tool implements the design created by the design tool, including executing storage system and host configuration commands and potentially migrating data to the appropriate devices.
- *solution monitoring tool*: once a data protection design has been deployed, this tool monitors the solution to determine whether the predicted behavior is realized and whether environmental failures occur as expected.
- *adaptive management*: the design tool, configuration tool, and solution monitoring tool can be used in concert to provide an iterative approach to adapt to changes in requirements or the environment [2].

3 Specifications for automating data dependability

In this section, we provide a brief overview of our approach’s declarative specifications for user requirements, failures and data protection techniques.

As described in [16], we believe it is necessary to distinguish between the following when specifying users’ data dependability requirements and data protection techniques:

- the *content* we are trying to protect (called *data*)
- the *access patterns* to the content (called *streams*)
- the *containers* in which data resides (called *stores*)

Distinguishing between data and stores allows us to separate the requirements of the content from the properties of the container. Data descriptions include attributes such as capacity, data loss rate, retention and expiration times, and recovery dependencies; they are described more in Section 3.1.1. Store properties, which are provided by the underlying data protection techniques, are described in more detail in Section 3.3.

Describing data separately from streams allows us to separate reliability from performability. Our primary technique for specifying data accessibility is the use of stream performability requirements, which indicate how often certain performance levels should be achieved [17]. These stream requirements are described in more detail in Section 3.1.3.

Figure 3 (in the appendix) is an example that folds together many of the points we describe here. It is structured using the Rome data model [16]. The example

represents the format that would be seen by one of our design tools, rather than the form seen by people specifying this information, for which user-oriented GUIs are more appropriate. We encourage the reader to follow along as these concepts are introduced.

3.1 Specifying user requirements

In this section, we discuss the requirements for which the users must provide input, including data and stream attributes.

3.1.1 Data: content

Data is the information content that is stored. At the storage system level, a data item is relatively large – a logical volume, a complete file system, or a database table. Higher-level software, such as a file system or database, maps smaller data objects such as files or records into these larger-scale data objects. We assume the existence of a primary copy of a data item, possibly with completely or partially redundant secondary copies.

Reliability is the most important data property: how much data loss is tolerable? Although users are likely to answer “none,” they must decide how much they are willing to pay to bound the amount or rate of data loss. For a large system, it makes sense to think of a “mean data loss rate” that is achievable for a given price.

Different types of data may have different reliability needs. For instance, it might be more cost-effective to allow up to 30% of the data for an Internet search application to be lost than to pay for its protection, as this will only impact the quality of answers it can provide, without impacting its ability to provide an answer [6]. Similarly, a database index can be rebuilt from the raw data, so it can be protected with less expensive techniques than the data it indexes, as queries can be posed against the main table(s) temporarily.

Thus, we believe that the appropriate data-related dependability properties are as follows:

- *capacity*: the size of the content, in bytes.
- *dataLossRate*: the allowed rate at which a particular size of data loss can occur, specified as a <bytes, time interval> tuple. When the size is one byte, the interval is the inverse of the traditional “mean time to data loss” metric. Some failure modes and store designs are such that only a portion of the data may be lost (e.g., at most a day’s worth of updates). This measure allows different designs to exhibit different properties, even if their traditional “reliability” values are the same.
- *dependsOn*: data-level recovery or integrity dependencies. For instance, the indices for an order-entry database depend on the underlying tables, implying that the tables should be recovered first after a failure. Additionally, applications may share data, making the description of these dependencies a DAG rather than a tree.

3.1.2 Retrieval points

Keeping read-only copies of a data item's state is a classic technique in data protection. These copies have been called versions, generations, snapshots, checkpoints or backups; we prefer the term *retrieval points* to separate the intent from the technique. A retrieval point permits "time-travel" in the storage system by capturing the state of a data item at some moment, with the expectation that this state can be accessed in the future. Retrieval points can be used to satisfy many needs, including protection against device failure, protection against user error or malicious actions [12, 14], protection against software errors or data corruption, legal requirements (e.g., audits), preserving a particular data state or a related set of data item states (e.g., archiving all the designs for a particular aircraft engine), and wanting to look at prior versions "in case they still have value" (e.g., old RCS versions or the file system structure used last year).

Retrieval point properties depend on the purpose of the retrieval point. Some retrieval points are single-shot archives (e.g., the design for an airplane engine). Such an archive may have required retention times, and accessibility and reliability properties. Some retrieval points are better thought of as a series of related point-in-time snapshots, generated by some automatic technique. A series may be better described by the number of retrieval points it should contain and a bound on the intervals between them, which implicitly specifies how many recent updates the user is willing to lose.

The useful lifetime of a retrieval point depends on the time to discover the need to recover data. For example, a person may take a few seconds to realize that he or she made a mistake, or a file system consistency checker may only be run once an hour. This time-to-detect acts as a natural lower bound on how long an associated retrieval point should be kept. Often, time-to-detect is quite short: research indicates that humans typically catch about 70% of their own errors, often within a few minutes of making them [11].

We describe retrieval point properties as follows:

- *capacity* and *dataLossRate*: defined as for a data item. It is useful to distinguish between the *dataLossRate* for the retrieval point and that of the parent data item, because people may value these versions differently.
- *retention* and *expiration times*: how long the content must be retained, and when it must be expunged. For example, retain the data forever and never expunge it, or keep it for seven years and then discard it immediately. The expiration date must be no earlier than the retention date. Associating these times with a read-only retrieval point avoids the difficulties of deciding what to do for data items that are being updated, where the starting point of the retention/expiration period is often unclear: is it the creation time, the update time, or the last access time?
- *count*: the number of retrieval points of this type to retain. This attribute can implicitly specify how long

retrieval points should be kept. Note that the user cannot specify both a count and a retention time.

- *interval*: the interval between retrieval points. This attribute is a measure of the amount of data loss that can be tolerated on a failure or mistake. It can be specified as a time (e.g., 30 minutes), an amount of data (4MB), or a count of updates (1 million writes).

A data item can have a series of retrieval points, each with associated resiliency and lifetime properties. For example, specifying the three retrieval point series `<interval=60 seconds, count=60, dataLossRate=1 B / 5 minutes>`, `<4 hours, 2, 1 B / 12 hours>`, `<3 months, 1 B / 50 years>` might cause the system to take a low-resiliency snapshot every minute, a more resilient one every four hours, and a nearly indestructible one once every quarter.

3.1.3 Streams: access patterns

Streams describe the access patterns to an associated data item. Performance requirements may be as simple as request rates and request sizes, or rich enough to include other quantities, such as spatial and temporal locality, phasing behavior, correlations between accesses to different parts of the storage system, and response time goals. (A more detailed description can be found in [16].)

We specify data accessibility using stream-based performability requirements, which indicate how often certain performance levels should be achieved. In particular, we describe both the baseline performance requirements under normal operation and the performance requirements needed during *outages*, which are periods of degraded operation. The allowed frequency and duration of outages can also be specified.

This approach is in contrast to the traditional, rather simplistic, "number of nines" availability metric, which implicitly supports only "all" or "none" as its two performance levels. In our scheme, the traditional "no availability" case is an outage with zero access performance. In addition, several intermediate outage levels can be described, each with its own performance specification. For example, if a system can tolerate operating at 50% of normal performance for a while, then this time can be used to bound the recovery time for a mirrored disk failure.

We describe outages using the following attributes:

- *outage duration*: the time duration of the longest tolerable outage. Note that this quantity implicitly includes the time to detect, diagnose, repair and recover from the failure, as described in Section 3.2. As a result, the overall recovery time must be strictly less than or equal to the outage duration.
- *outage frequency*: the maximum number of separate outages that are permitted (or measured) during a user-specified period (e.g., a year).
- *outage fraction*: the fraction of the total time (averaged over the user-defined period) that can be outages.

A common approach to handling the case where data has to be retrieved from a secondary copy is the use of a

“time to first byte” specification, to cover the recovery time. We believe that this case is better handled by specifying the allowed delay as an outage, which can also be used to put bounds on the number and frequency of such recoveries.

A final category of stream characteristics governs the amount of recently-written data that may be lost on a failure (e.g., from a file system write buffer in volatile memory). This amount is related to the update rate and the fraction of the data that is changing, so we believe these notions to be most appropriately categorized as stream (rather than data) characteristics:

- *recent-write data loss*: the amount of the most-recently-written data that can be lost on a failure. This parameter may be specified in terms of time (e.g., no more than the last 30 seconds) or in terms of volume (e.g., no more than 1MB).
- *recent-write loss frequency*: an occurrence frequency can be associated with this event to bound how often it may happen.

Note that individual retrieval points probably have different stream characteristics from each other and from the parent data item. Applications may only update the data item (e.g., the most recent copy), while they may read either the most recent copy or one of the retrieval points. For example, a retrieval point implemented as a file system snapshot may be read by the backup system to copy the data to archival media. Alternately, a retrieval point may be read to restore data upon software or user error. A retrieval point may also be directly accessed by an application. For example, a data mining algorithm may be run on a retrieval point for data from the OLTP database. The characteristics of such streams will have a large impact on what techniques can be used to provide the retrieval point (e.g., rapid recovery from user error and tape backup are not a good match).

3.1.4 Common attributes

Several attributes cut across the data and stream attribute categories, including:

- *application*: the business functionality that corresponds to a particular data item or stream (e.g., order-entry OLTP database or email server). The application is used to group related data items and streams.
- *dependsOn*: application-level recovery dependencies or ordering requirements. For instance, after a site failure, the OS data should be recovered before any of the application data.
- *utility*: Trade-offs can be specified using utility functions [8]: utility is positive when the system is delivering value (e.g., performing at or above its goal), and negative (e.g., a penalty) when the system under-achieves the goals. Such a utility function would allow us to prioritize applications differently. For example, getting 100 requests/sec for the order entry data may be more important than getting any requests to email data, but that once email achieves

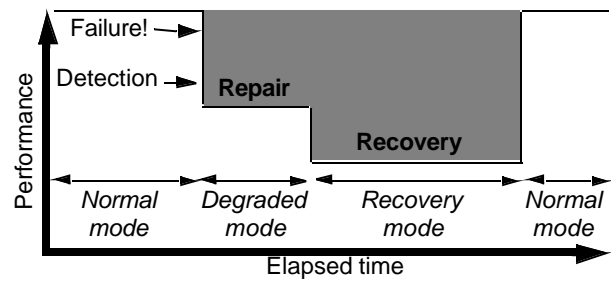


Figure 2: failure behavior modes and performance.

50 requests/sec, then increasing the order-entry capability would be valuable.

The reader can think of a utility function being provided for any of the above metric values. It is important that there is a “common currency” for utility; ultimately, we believe that this should be the same as the currency used to pay for the system. We are currently considering how best to specify generalized utility functions.

3.2 Specifying storage system failures

Failures can cause a system to lose data or to lose accessibility to data. They may occur at many points in the system, including the hardware, the software, and the humans interacting with the system. In fact, the literature suggests that humans are now responsible for the largest fraction of failures in many systems [11].

Our model of failure behavior [17] is illustrated in Figure 2. A *failure* occurs; some (hopefully short) time later this is *detected* and *diagnosed*, and the system enters *degraded mode* operation. If the technique(s) mask the failure, operation will continue at a (potentially reduced) non-zero performance level. After the failure is repaired, *recovery* is then initiated, and the system returns to normal operation (which may or may not be identical to the initial state, due to load balancing issues, etc.)

We characterize failures by the likelihood of their occurrence and the severity of their effects:

- *failure frequency*: what is the likely rate of failures of this type (expressed as an annual failure rate per entity)?
- *failure scope*: what part of the system is affected by the failure? The scope includes how many objects are affected at a time (e.g., all disks of a particular type or all files in a directory). The design system itself will calculate the effects of cascading, dependent failures.
- *failure correlations*: how often are failures not independent? For example, multiple disks may fail at the same time if the problem is excessive temperature.
- *failure manifestation*: how does the failure manifest itself? Possibilities include fail-stop (our focus for now), fail-stutter [4] or Byzantine failures.
- *early warning*: how much warning time is provided

before the failure? For true failures, this amount is likely zero; however, planned maintenance operations may provide as much as weeks of warning.

- *failure duration*: how long is the failure expected to last? For planned maintenance, we may be able to estimate the duration of the outage.
- *failure fix unit*: what's the minimum field replaceable unit (FRU) to fix this failure? For example, if a disk fails, it is more cost-effective to replace the defective disk than to replace the entire array. Alternately, if a SCSI bus fails, an entire disk array enclosure may need to be replaced. The failure fix unit will ultimately determine how much data must be recovered.

Examples of hardware-oriented failure scopes include:

- *components* (e.g., sector, disk, controller, cache memory, link, bus, UPS, fan, cable, plug)
- *subsystems* (e.g., array, host, switch, LAN, air conditioning unit, building power transformer)
- *racks* (e.g., a set of subsystems)
- *rooms* (e.g., a set of racks)
- *buildings* (e.g., a set of rooms)
- *sites* (e.g., a set of buildings)
- *areas* (e.g., a set of sites, city, earthquake zone)

Software- and user-oriented failures have similar kinds of scopes: a file, all files owned by a user, a file system, a logical volume, an operating system partition, a cluster file system, etc. Similar scopes exist for a database: record, table, table-space, and the entire database.

An alternate approach to failure scopes and correlations is to specify dependence relationships between components, which can be used to construct hierarchical fault trees [13]. We are currently investigating which alternatives are most appropriate.

We assume that initial failure estimates are provided to the storage design system by the provider of the hardware or software components. This information may be augmented by monitoring the operation of a deployed system.

3.3 Specifying data protection techniques

Stores are containers in which data items reside. The family of containers used to store data is organized as a hierarchy: host logical volumes reside on one or more disk array logical units, which are comprised of one or more disks, etc. Stores may employ different data protection techniques, which use redundant data representations to ensure that data is not damaged if something fails.

Different techniques provide a wide range of performance, failure tolerance and cost properties. They differ in their:

- baseline performance under normal operation
- degraded- and recovery-mode (i.e., outage)

performance

- ability to tolerate different failure properties
- repair and recovery costs (e.g., time, money)
- cost of implementation (e.g., space, time, money)

Selecting the best data protection techniques to satisfy user requirements requires understanding these properties for each candidate technique. We assume this information will be supplied by the provider of the data protection technique, estimated using models, or calculated by the storage design system, as described below.

To capture the performance categories, the design system needs one or more *performance models* of the technique's behavior – typically, one for each mode in which it can operate (normal, degraded, recovery). Several techniques have been described in the literature for estimating storage device performance under normal mode quickly and efficiently enough to be used in a storage design tool (e.g., [15]).

The design system needs to determine the probability of each performance mode for each technique. This requires knowledge of:

- *failure tolerance*: which failure scopes can the technique handle? We believe that this is best thought of as the set of performance modes entered for each associated failure scope.
- *data loss*: how much data is lost for a failure scope? Zero means that the technique masks or tolerates the failure.
- *repair time*: how long does repair take, before recovery can commence? This time includes physical repair, and can be short if the system includes hot spares for the failed component.
- *data to be reconstructed*: this value will be computed from the failure fix unit for a particular failure scope. For instance, if an entire disk enclosure must be replaced, all disk array logical units that use disks in that enclosure must reconstruct data.
- *recovery time*: this value will be calculated as a function of the desired performance levels and tolerable outages. For example, if recovery is achieved by copying data, then the speed of recovery can be varied to control the amount of disruption to the foreground load. Working backwards from the user's maximum allowed outage duration and frequency allows us to calculate the recovery traffic that will achieve recovery within the tolerable outage bounds. This traffic will have the least effect on the foreground load. Whether the resulting foreground performance is adequate is then a function of the workload requirements. If not, then the design system must choose some other technique to provide adequate protection and performance.

We are still in the early stages of mapping this space, and expect to expand this specification framework as we gain more experience with how the specifications, performance models and design tools interact.

4 Related work

Data dependability is a vast space. Much of the existing work in the systems community is on devising new data protection techniques, rather than on providing guidance on how to choose between them. CMU's PASIS project is trying to understand trade-offs in the design of survivable storage systems [18]. Their focus is primarily on threshold coding schemes and cryptographic techniques to increase data dependability (including availability, reliability and security) in the wide area.

Additional work in the systems literature deals with evaluating the performability of a system. Wilkes and Stata [17] propose a method for describing performability using variations in quality of service under normal and degraded modes of operation. Brown and Patterson [7] describe how to measure the availability of RAID systems using a similar performability framework.

In the performance space, we have successfully used the approach of declarative goal specification [16] to automate the mapping of performance and capacity requirements onto storage designs [1, 2, 3].

Although the system administration community chooses between different data protection techniques regularly, there is little published literature on the topic. The dependable systems community has developed a vocabulary for describing dependability and failure behavior [5] and techniques for modeling various aspects of dependability [9] [13]. We are in the process of understanding the considerable body of literature from this community.

5 Conclusions

Increasing data dependability is an important problem, whose value will continue to increase as service dependability becomes more important. The proliferation of techniques, the complexity of interactions and the richness of demands are making purely manual methods cumbersome. We believe that automating the selection of techniques is both necessary and promising, and that the correct goal is a broadly-scoped automated design tool for data protection. This paper has outlined our approach, and described the data model we use to describe user requirements, failure characteristics and data protection technique behaviors.

We see a number of areas of future work. First, how do we intuitively ask users what they want? Second, how do we build a suite of design tools that automates the selection of data protection techniques? Finally, we challenge developers of new techniques to describe their behavior under different operational modes.

6 Acknowledgements

The authors thank Eric Anderson, Christos Karamanolis, Mahesh Kallahalla, Ram Swaminathan, and Jay Wylie for their valuable insights and comments on earlier versions of this paper.

7 References

- [1] G. Alvarez, et al. "Minerva: an automated resource provisioning tool for large-scale storage systems," *ACM Transactions on Computer Systems*, 19(4):483-518, November 2001.
- [2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. "Hippodrome: running circles around storage administration," *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, January 2002, pp. 175 - 188.
- [3] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. "Ergastulum: an approach to solving the workload and device configuration problem," HP Laboratories SSP technical memo HPL-SSP-2001-05, May 2002.
- [4] R. Arpaci-Dusseau and A. Arpaci-Dusseau. "Fail-stutter fault tolerance," *Proc. of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001, pp. 33 - 38.
- [5] A. Avizienis, J.-C. Laprie and B. Randell. "Fundamental concepts of dependability," *Proc. of the 3rd Information Survivability Workshop*, October 2000, pp. 7 - 12.
- [6] E. Brewer. "Lessons from giant-scale services," *IEEE Internet Computing*, 5(4):46-55, July 2001.
- [7] A. Brown and D. Patterson. "Towards availability benchmarks: a case study of software RAID systems," *Proc. of the 2000 USENIX Annual Technical Conference*, June 2000, pp. 263 - 276.
- [8] G. Candea and A. Fox. "A utility-centered approach to dependable service design," *Proc. of the 10th ACM-SIGOPS European Workshop*, September 2002.
- [9] B. Haverkort, R. Marie, G. Rubino and K. Trivedi, eds. *Performability modeling: techniques and tools*, John Wiley and Sons, Chichester, England, May 2001.
- [10] K. Keeton and E. Anderson. "A backup appliance composed of high-capacity disk drives," *Proc. of HotOS-VIII*, May 2001, p. 171.
- [11] D. Patterson. "A new focus for a new century: availability and maintainability >> performance," Keynote speech at *USENIX FAST*, January 2002. Available from <http://www.usenix.org/publications/library/proceedings/fast02/>.
- [12] D. Santry, et al. "Deciding when to forget in the Elephant file system," *Proc. of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, December 1999, pp. 110 - 123.
- [13] D. Sieworek and R. Swartz. *Reliable computer systems: design and evaluation*, A. K. Peters, Third Edition, 1998.
- [14] J. Strunk. et al. "Self-securing storage: protecting data in compromised systems," *Proc. of Operating Systems Design and Implementation (OSDI)*, San Diego, CA, October 2000, pp. 165-180.
- [15] M. Uysal, G. Alvarez and A. Merchant. "A modular, analytical throughput model for modern disk arrays," *Proc. of the 9th Intl. Symp. on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS)*, August 2001, pp. 183 - 192.
- [16] J. Wilkes. "Traveling to Rome: QoS specifications for automated storage system management," *Proc. of the Intl. Workshop on Quality of Service (IWQoS)*, June 2001, pp. 75 - 91.
- [17] J. Wilkes and R. Stata. "Specifying data availability in multi-device file systems," *Proc. of the 4th ACM-SIGOPS European Workshop*, September 1990; published as *Operating Systems Review* 25(1):56-59, January 1991.

[18] J. Wylie, et al. "Selecting the right data distribution scheme for a survivable storage system," Technical report CMU-CS-01-120, Carnegie Mellon University, May 2001.

8 Appendix: example

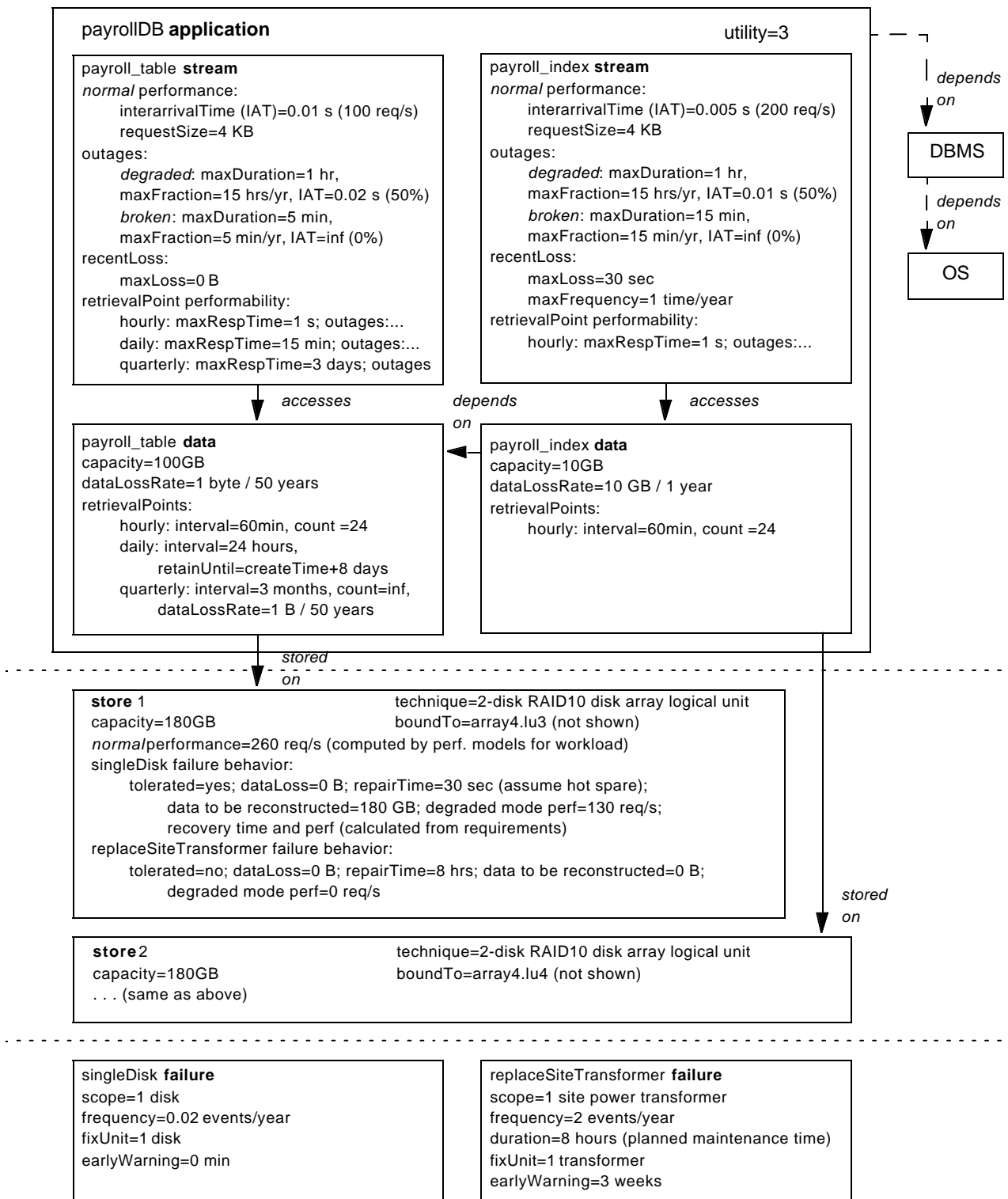


Figure 3: a (simplified) sample data dependability specification. A payroll database contains both a table and an index. In normal mode, the table gets 100 requests/sec; in “degraded” mode, it can operate at half that rate for an hour at a time; and it can be “broken” no more than 5 minutes a year (“five nines availability”). The index is accessed at twice the rate. Three retrieval points are defined for the table, with different time intervals and retention periods. The data is stored on RAID10 disk array logical units, whose performance under normal mode and various failure conditions is provided by performance models. We note that these stores tolerate the single disk failure, but not the site transformer replacement.

Session 6. Operating System's Structures

1. Nooks: An Architecture for Reliable Device Drivers

Authors: Michael M. Swift, Steven Martin, Henry M. Levy, Susan G. Eggers

page 101

2. Sub-Operating Systems: A New Approach to Application Security

Authors: Sotiris Ioannidis, Steven M. Bellovin, Jonathan M. Smith

page 108

Nooks: An Architecture for Reliable Device Drivers *

Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195, USA

{mikesw, stevaroo, levy, eggers} @cs.washington.edu

1 Introduction

With the enormous growth in processor performance over the last decade, it is clear that reliability, rather than performance, is now the greatest challenge for computer systems research. This is particularly true in the context of Internet services that require 24x7 operation and home computers with no professional administration. While operating system products have matured and become more reliable, they are still the source of a significant number of failures. Furthermore, recent studies show that device drivers are frequently responsible for operating system failures. For example, a study at Stanford University found that Linux drivers have 3 to 7 times the bug frequency as the rest of the OS [4]. An analysis of product support calls for Windows 2000 showed that device drivers accounted for 27% of crashes, compared to 2% for the kernel itself [16].

The reasons for the high rate of driver failures are four-fold. First, drivers are typically written by device manufacturers rather than by operating system developers with extensive kernel programming experience. Second, drivers are frequently created by copying and editing code templates from existing drivers, often without complete understanding, leading to subtle bugs. Third, the kernel programming environment has many unenforced or poorly-documented conventions about synchronization and memory access, making kernel-mode programming and debugging challenging, at best. Finally, driver programming often requires understanding the operation of complex asynchronous devices, their control protocols, and their failure modes. As the number of new devices available increases to support new applications, such as cam-

eras, digital video, etc., so does the number of drivers required and the number of (relatively unskilled) programmers responsible for creating them.

Device drivers can be viewed as a type of kernel extension, added after the fact. Commercial operating systems are typically extended by loading unsafe object code and linking it directly with the kernel. There have been many attempts to solve the general problem of safely extending the kernel [6, 2, 20], but they have demanded that programmers change the way in which they write code or the way in which operating systems are structured. Such approaches are unworkable for device drivers, which are the most common operating system extensions and represent a huge investment in development time; hence none of these approaches have been successful in a commercial system.

System availability depends not just on fault isolation, but also on quick recovery from faults. Therefore, the operating system must not just isolate faulty device drivers, but also allow them to quickly resume service, either after restarting or after recovering previous actions in progress. Recovery is also increasingly important due to the rising problem of hardware failures [16], which depends on isolating the effects of a fault and quickly recovering to a pre-fault state.

In the future, it is clear that improving operating system reliability depends on improving device driver reliability, because the kernel is no longer the primary source of bugs (or kernel-mode code!). In addition, as software matures and device integration levels shrink, hardware failures will become a greater problem. As a result, operating systems need to provide support for (1) tolerating and recovering from faulty drivers, and (2) tolerating and recovering from faulty hardware. The NOOKS project is examining mechanisms and architectures to meet these goals.

*This work was supported in part by the National Science Foundation (grants ITR-0085670 and CCR-0121341).

2 Approaches

Many approaches have been proposed to safely execute user- or kernel-mode code, including device drivers. One difference between safely executing drivers and safely executing general kernel extensions is that one can assume that most device drivers are trustworthy: the problem is one of safety and not security, and absolute safety may not even be needed. Table 1 shows five key hardware and software techniques that can isolate driver code from the OS kernel. Each of these techniques has benefits and drawbacks, and may be appropriate in certain situations. Table 1 also shows the systems that used each technique.

Table 2 shows the relative advantages and disadvantages of each approach along the axes of software engineering, performance for large and small volumes of data, and ability to isolate memory corruption and deadlock errors.

In more detail:

1. Kernel wrapping surrounds all calls into and out of device drivers with special code, allowing resources to be tracked and pre- and post-conditions to be verified. Kernel wrapping can ensure that memory not owned by the driver is not freed, and that interrupts are enabled before blocking. However, kernel wrapping cannot prevent a driver from accidentally corrupting the operating system by writing through a stray pointer.
2. Virtual memory protection can be used to isolate data corruption problems, which are one of the most common driver faults, but can't catch deadlock errors, such as those caused by improper disabling of interrupts.
3. Lowering the privilege level of drivers (e.g., to supervisor or user level) prevents them from executing privileged instructions, accessing privileged address space, and corrupting the kernel. However, there is a large performance penalty, because calls into drivers require an additional trap and return to change privilege level.
4. Software fault isolation (SFI) [23] provides many of the benefits of a privilege level change, but is difficult to implement when the range of addresses accessible are not contiguous. In contrast to lowering the privilege level, it is very cheap to call into and out of SFI code, but SFI code executes more slowly. In addition, SFI does not easily support recovery in the form of copy-on-write, which hardware memory protection does support.

5. Finally, safe languages such as Java [7] and Modula-3 [17] can prevent drivers from uncontrolled access to kernel memory. However, using a safe language requires rewriting drivers and may introduce significant overhead when safely copying data in and out of the driver. There is also not as of yet a good mechanism for accessing a device in a type-safe fashion.

As a result, there is no single approach that is applicable for all device drivers. High performance devices, such as network and disk interfaces, require minimum overhead for large quantities of data to match the speed of the device and must run in the kernel. Low performance devices, though, such as keyboards, mice, and serial devices, do not always require the performance benefit of running in the kernel with no protection. The reliability needs of computer installations also vary: a bank may be willing to suffer a significant performance decrease in order to improve reliability, while a game player would risk crashing periodically to improve the realism of the game. Thus, it is important for the operating system to support a variety of techniques, so that driver writers and system installers can choose the most appropriate for them.

Beyond fault isolation, fault recovery is also increasingly important. Some techniques, such as memory protection, lend themselves toward automatic fault recovery. For example, Discount Checking [13] and Lightweight Recoverable Virtual Memory [19] use copy-on-write to maintain a shadow copy of all memory accessed by an application, allowing automatic recovery when a fault is detected. Other techniques, like safe languages and SFI, provide little support for recovery.

3 Nooks Architecture

We propose that operating systems should support executing drivers in a fault-isolating and recoverable environment so that a faulty driver cannot prevent the rest of the OS from functioning. In addition, the operating system should offer multiple levels of isolation and performance to reflect different driver needs. Our goal is to provide these features with only an incremental change to the operating system and device driver architecture, to maximize compatibility with the existing code base.

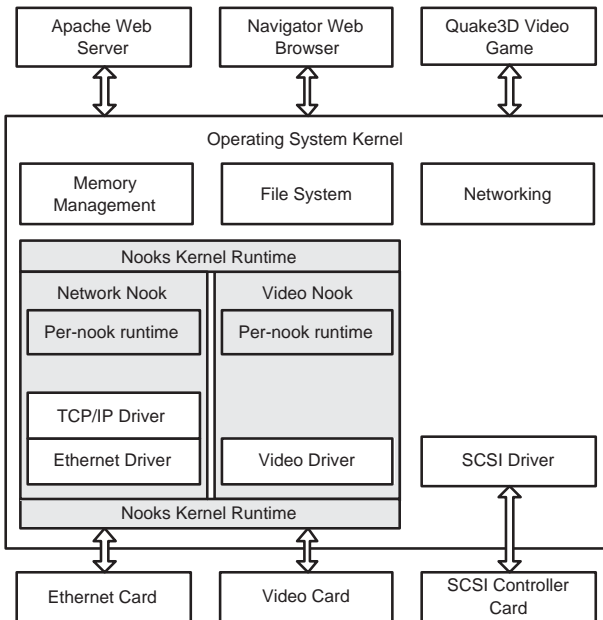
Figure 1 shows our proposed architecture, called NOOKS. A *nook* is a protected environment for driver execution. Not all devices execute within a nook, as illustrated by the SCSI device driver in the figure. In addition, multiple drivers may execute within the same nook for performance reasons, as illustrated by

Table 1. Kernel extension safety approaches

Name	Description	Where Used
<i>Kernel Wrapping</i>	Verify all parameters on calls between the kernel and device drivers	Microsoft Driver Verifier [15]
<i>Hardware Memory Protection</i>	Prevent device drivers from writing to kernel memory	Palladium [3], Shinagawa [21]
<i>Privilege Level Change</i>	Prevent device drivers from executing privileged instructions and/or emulate privileged instructions	L4 [12], Exokernel [6]
<i>Software Fault Isolation</i>	Inject code into device drivers to ensure that addresses and instructions are safe	Vino [20]
<i>Safe Languages</i>	Rely on the compiler/virtual machine to allow only safe (non-faulting) drivers to be loaded	SPIN [2]

Table 2. Comparison of driver safety approaches

	<i>Kernel Wrapping</i>	<i>Hardware Memory Protection</i>	<i>Privilege Level Change</i>	<i>Software Fault Isolation</i>	<i>Safe Languages</i>
<i>Requires rewriting driver</i>	No	No	No	Maybe	Yes
<i>Easily supports recovery</i>	No	Yes	Yes	No	No
<i>High performance for small data volumes</i>	Yes	No	No	Yes	Yes
<i>High performance for large data volumes</i>	Yes	Yes	Yes	No	No
<i>Isolates memory corruption</i>	No	Yes	Yes	Maybe	Yes
<i>Prevents most deadlocks</i>	Maybe	No	Yes	Yes	Yes

**Figure 1.** NOOKS architecture diagram

the combination of the Ethernet and TCP/IP drivers within the *network* nook. Nooks interpose between devices and device drivers by forwarding interrupts and, depending on the level of safety required, emulating access to memory-mapped device registers. Nooks also wrap calls from the operating system kernel into device drivers and from device drivers into the kernel, allowing the operating system to track resource usage and verify data that is passed into and out of the kernel. Rather than fully isolate device drivers in a separate address space, all drivers execute in the kernel address space, but within different protection domains. Thus, a device driver may use pointers supplied by the kernel without copying the data or translating addresses. However, the NOOKS architecture prevents device drivers from writing to memory outside their protection domain, limiting the damage of an errant memory access. Initially we use virtual memory protection and lowered privilege levels for isolating and recovering faulty code, but we plan to experiment with Software Fault Isolation (SFI) as well.

The NOOKS architecture minimizes the number of

crossings between the kernel protection domain and device drivers by separating kernel resources into those that must be shared from those that are only shared incidentally. For example, the processor must be shared among all drivers, so kernel intervention is required for scheduling functions. However, other operating system resources, such as wait queues and memory heaps, are only shared for convenience. NOOKS takes advantage of these two classes of resources to improve performance by duplicating the resources that are only incidentally shared. Drivers may then directly access the resource without crossing to the kernel's protection domain. In addition, some of the work that must be performed in the kernel need not be performed synchronously, such as delivering network packets. These operations can be deferred until the driver has completed execution, and then performed in a single batch.

Device drivers may require small changes to execute within the NOOKS architecture. In particular, operating system support routines that make kernel data structures directly available to device drivers (such as by returning a pointer to an kernel data structure which the driver then updates) cannot be supported. Instead, drivers must call wrapper routines that update kernel data structures on their behalf. The NOOKS architecture will initially support two levels of recovery: full restart, which unloads and restarts drivers, and rollback, which uses recoverable virtual memory to maintain a shadow copy of driver state, allowing it to be recovered after a fault. Many device requests are by their nature idempotent, such as sending or receiving a network packet or writing disk block, so in some cases a failed operation can be retried safely.

4 Implementation

We implemented a prototype of the NOOKS architecture in the Linux kernel. The prototype runs on Linux version 2.4.10, and we have experimented with isolating network interface device drivers, including the 3Com 3c905 fast ethernet adapter and the Intel Pro/1000 gigabit ethernet adapter. The prototype follows the architecture shown in Figure 1, and wraps all calls into and out of device drivers to prevent write-sharing of data and to verify parameters. We plan on using hardware memory protection to isolate drivers, and we therefore maintain a copy of the kernel pagetable for drivers that only grants read access to kernel text and data.

While our implementation does not execute drivers in a separate protection domain, it does emulate the cost of switching protection domains. First, to emulate the cost of changing page tables, we flush the TLB on

all calls into or out of a driver. Second, to emulate the cost of lowering the privilege level of a driver, we execute a software trap and return both when entering and leaving a driver as well as when the driver executes a privileged operation, such as disabling interrupts.

Our prototype NOOKS implementation wraps 147 calls made by device drivers into the Linux kernel, and 103 calls from the kernel into device drivers through ten different interfaces. To ensure that drivers do not call directly into kernel functions, we modified the *insmod* program, which is responsible for binding symbols in dynamically loaded kernel code, to bind symbols imported by these drivers to the NOOKS wrappers rather than to the kernel functions.

The Linux operating system and the Intel IA-32 architecture proved to be difficult choices for implementing NOOKS. First, Linux allows drivers to modify many kernel data structures. For example, drivers typically update a kernel queue when freeing network packets. In addition, many kernel functions are implemented as inline function calls. We converted several functions to true procedure calls, which requires that drivers be recompiled to execute within our prototype. The Intel IA-32 architecture proved difficult because of its hardware TLB miss handler. The architecture enforces a page table layout that does not support multiple protections for the same page at the same privilege level. As a result, we instead duplicated the kernel page table and copied all updates from the kernel page table.

The NOOKS wrapper code copies all parameters passed from drivers into the kernel, and also verifies that pointers reference data structures allocated to the driver making the call. In addition, the wrapper layer maintains a mapping between kernel data structures and copies of the data structure used by the driver, and ensures that the data structures are synchronized by copying changes during calls into the driver or kernel.

We emulate different levels of protection, ranging from just wrapping calls to executing drivers at a lower privilege level by controlling the cost of protection domain switches and privileged operations. We are therefore able to determine the impact of different protection techniques on specific device drivers and workloads.

5 Performance

To test the performance of our prototype, we used the Netperf benchmarking tool [9] to measure TCP bandwidth and UDP request/response performance. The bandwidth tests stream 32KB messages between two machines, and the UDP tests send a 100 byte request and receive a 200 byte response. Our experimental

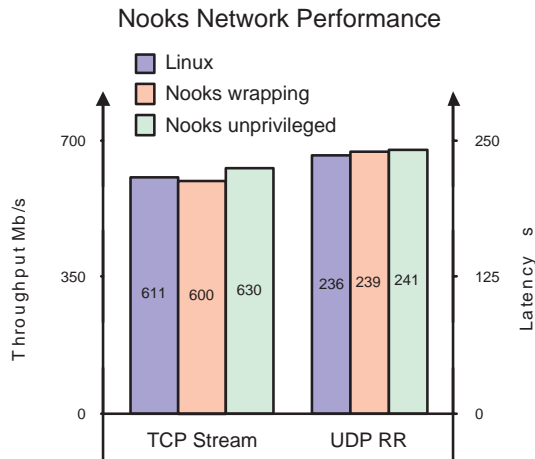


Figure 2. NOOKS Netperf performance compared against unmodified Linux

platform was a pair of PCs with 1.7 GHz Pentium 4 processors, 1 GB of memory, and gigabit Ethernet adapters, one of which ran drivers isolated with NOOKS. We tested Linux with the driver loaded as a module, and NOOKS with just wrapping and with full protection, which entails both flushing the TLB and executing a software trap on all calls into and out of the driver and on privileged instructions. In all cases we used the default values for the driver's parameters.

The results are shown in Figure 2, and demonstrate that isolating a driver using NOOKS has a negligible impact on network performance. Despite the additional overheads from flushing the TLB and executing a trap, the bandwidth of TCP streams actually improved, from 611 Mb/s to 630 Mb/s. We believe the performance increase is due to the increased number of packets received during each driver interrupt, which increased from 7.1 to 8.6 packets per interrupt. The UDP request/response test measures the round-trip time across the network and through the network stacks on the two machines. The latency increased slightly, from 236 microseconds to 241 microseconds, demonstrating the low overhead introduced by NOOKS.

We measured the impact of isolating the network driver on interrupt handling by measuring the number of cycles spent handling interrupts from the network interface. On Linux, the Intel PRO/1000 driver executes in 20,800 cycles on average. With just wrapping, handling an interrupt takes nearly twice as long, 37,200 cycles. Raising the privilege level and changing page tables raises that cost to 47,800 cycles, about 15 microseconds longer than plain Linux. We also measured

the load incurred by processing the TCP stream. On unmodified Linux, processing TCP required 17.6% of the CPU, while running the driver isolated with both wrapping and a lowered privilege level required 20.7% of the CPU, an 18% increase.

Overall, these experiments demonstrate that on modern processors the cost of isolating device drivers is low. Furthermore, other architectures with faster operating system operations, such as the Alpha [11] or Itanium [5], could further reduce these overheads.

6 Related Work

Many projects have tackled the difficulty of writing device drivers. The Stanford study using the MC tool [4] and Microsoft's SLAM project [1] mechanically found many bugs in device drivers, but did not automatically fix those bugs. Microsoft's Driver Verifier [15] wraps operating system calls, but is meant as a debugging tool only, and does not prevent memory corruption. The Devil Project [14] aims to simplify the process of writing device drivers by providing a domain-specific language for specifying the interface between the device and the processor, which could be used as part of NOOKS to better isolate driver's hardware access. The WinDriver architecture [10] and Hunt's user-mode drivers [8] both allow device drivers to be run in user-mode, but support a different API from the kernel and don't provide the performance option of executing in the kernel but with memory protection. Van Maren [22] built a user-mode device interface for the Fluke microkernel, but it was not applicable to conventional operating systems. Finally, the Uniform Driver Interface project (UDI) [18] provides a driver interface to the kernel that prevents deadlock and allows for memory isolation, but again requires that drivers be completely rewritten.

7 Conclusion

Reliable services depend on a reliable operating system. While the core operating system kernel code has become reliable, device drivers have not kept pace. Device drivers are by their very nature difficult to write and even worse, are not written by experts at kernel programming. Therefore, operating systems must assist device driver writers by reducing the penalty of a faulty driver. The NOOKS architecture accomplishes this goal by isolating drivers with a variety of techniques, including kernel wrapping, virtual memory protection, privilege level lowering, and software fault isolation, which are suited to many different environments

and driver types. NOOKS does not require rewriting drivers, and can support recovery from both software and transient hardware errors using copy-on-write virtual memory techniques to maintain a shadow copy of uncorrupted memory. Executing drivers within a nook isolate the two most common driver bugs, memory corruption and deadlock, leading to more reliable systems. Finally, the NOOKS architecture can achieve high performance for large volumes of data using virtual memory remapping, while also maintaining performance for low-bandwidth devices with software fault isolation.

References

- [1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. 29th POPL*, Portland, OR, Jan. 2002.
- [2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th SOSP*, pages 267–284, Copper Mountain, Colorado, Dec. 1995.
- [3] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proc. 17th SOSP*, pages 140–153, Kiawah Island Resort, South Carolina, Dec. 1999.
- [4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empiracle study of operating system errors. In *Proc. 18th SOSP*, Lake Louise, Alberta, Oct. 2001.
- [5] I. Corporation. *The IA-64 Architecture Software Developer's Manual*. Intel Corporation, Jan. 2000.
- [6] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th SOSP*, pages 251–266, Copper Mountain Resort, Colorado, Dec. 1995.
- [7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [8] G. Hunt. Creating user-mode device drivers with a proxy. In *Proc. 1997 USENIX Windows NT Workshop*, Seattle, WA, Aug. 1997.
- [9] R. Jones. Netperf: A network performance, version 2.1, 1995. Available at <http://www.netperf.org>.
- [10] Jungo. Windriver cross platform device driver development environment. Technical report, Jungo Corporation, Feb. 2002. <http://www.jungo.com/windriver.html>.
- [11] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.
- [12] J. Liedtke. On μ -kernel construction. In *Proc. 15th SOSP*, pages 237–250, Copper Mountain Resort, Colorado, Dec. 1995.
- [13] D. E. Lowell and P. M. Chen. Discount checking: Transparent, low-overhead recovery for general applications. Technical Report CSE-TR-410-99, University of Michigan, Nov. 1998.
- [14] F. M  rillon, L. R  veill  re, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proc. 4th OSDI*, pages 17–30, San Diego, CA, Oct. 2000.
- [15] Microsoft. Windows XP device driver development kit. Technical report, Microsoft Corporation, Oct. 2001.
- [16] B. Murphy. Fault tolerance in this high availability world. Talk given at Stanford University and University of California at Berkeley. Available at <http://research.microsoft.com/users/bmurphy/FaultTolerance.htm>, Oct. 2000.
- [17] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.
- [18] Project-UDI. Introduction to UDI version 1.0. Technical report, Project UDI, Aug. 1999. Available at http://www.project-udi.org/Docs/pdf/UDI_tech_white_paper.pdf.
- [19] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *Proc. 14th SOSP*, pages 146–160, Asheville, North Carolina, Dec. 1993.
- [20] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd OSDI*, pages 213–227, Seattle, Washington, Oct. 1996.
- [21] T. Shinagawa, K. Kono, and T. Masuda. Exploiting segmentation mechanism for protecting against malicious mobile code. Technical Report 00-02, Dept. of Information Science, University of Tokyo, May 2000.
- [22] K. T. Van Maren. The fluke device driver framework. Master's thesis, Department of Computer Science, University of Utah, Dec. 1999.
- [23] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. 14th SOSP*, pages 203–216, Asheville, North Carolina, Dec. 1993.

Sub-Operating Systems: A New Approach to Application Security

Sotiris Ioannidis
sotiris@dsl.cis.upenn.edu
University of Pennsylvania

Steven M. Bellovin
smb@research.att.com
AT&T Labs Research

Jonathan M. Smith
jms@dsl.cis.upenn.edu
University of Pennsylvania

Abstract

Users regularly exchange apparently innocuous data files using email and ftp. While the users view these data as passive, there are situations when they are interpreted as code by some system application. In that case the data become “active”. Some examples of such data are Java, JavaScript and Microsoft Word attachments, each of which are executed within the security context of the user, allowing potentially arbitrary machine access. The structure of current operating systems and user applications makes solving this problem challenging.

We propose a new protection mechanism to address active content, which applies fine-grained access controls at the level of individual data objects. All data objects arriving from remote sources are tagged with a non-removable identifier. This identifier dictates its permissions and privileges rather than the file owner’s user ID. Since users possess many objects, the system provides far more precise access control policies to be enforced, and at a far finer granularity than previous designs.

1 Introduction

Most classical work on computer security focuses on operating systems. One of the fundamental tasks of an operating system is resource allocation and control: making sure that different users have fair access to shared resources, such as disk space and CPU time, all the while ensuring that access restrictions are honored. A secure system is one where these controls are effective in the face of deliberate attempts at subversion.

The advent of ubiquitous networking has changed all this. While operating system security is still important, many of the new threats arise because of network activity. Often, these threats are data-driven, and within the confines of a single user’s protection boundaries. For example, pieces of mobile code, run with the permissions of some user, may attempt to steal or destroy files belonging to that user.

Naturally, the applications accepting the mobile code attempt to guard against such things. But their record has at best been mixed. The problem is more acute because many more network objects can, in some sense, be considered “code”, even if not intended as such.

Many of the security problems that have occurred have been due to problems in determining file permissions. Applications typically resort to pattern-matching, a dangerous and error-prone technique. For example, CERT Advisory CA-98.04 describes a problem on a Web server that didn’t properly check the so-called “short name” when the actual name of the file did not fit into the legacy 8.3 format. Similarly, Advisory CA-2000-15 describes how a Java applet could open `file:` URLs, thus reading files from the local machine. In the first case, the application misunderstood the operating system’s file name semantics; in the second, the check was omitted entirely.

Operating systems rarely have such problems. File permissions are associated with the *file* itself; any attempt to open the file will cause a permission check, regardless of how the file was accessed. OS security failures typically occur when a privileged program is tricked into opening a file; the kernel’s access control mechanism is simple enough that it is rarely at fault.

In this paper, we describe an architecture, called SubOS, for fine grain control of data objects. In SubOS data objects are treated as users (sub-users), with their own privileges and permissions. Figures 1 and 2 demonstrate the differences between a regular and a SubOS-enabled operating system. On a regular operating system user applications execute with the permissions of the user and have access to the underlying system identical to that of the user. Under SubOS, applications execute with the permissions of the data object (sub-user) they operate on. This allows for finer grain control, and therefore greater protection from malicious data objects.

The paper is organized as follows. In Section 2 we discuss the motivation behind this work. In Sections 3 and 4 we present the design and implementation details of a SubOS-capable OpenBSD [2] system, and two applications that benefit from such an architecture. In Section 5 we dis-

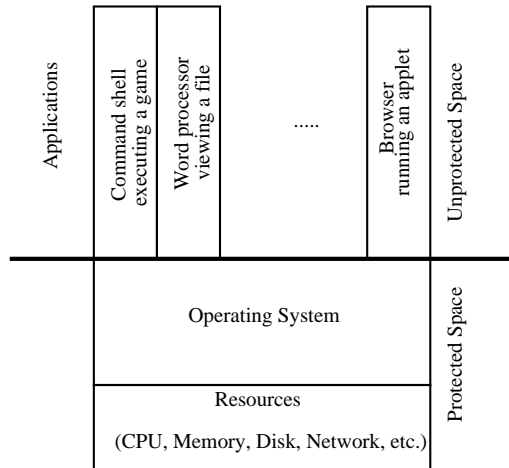


Figure 1. User applications executing on an operating system maintain the user privileges, allowing them almost full access to the underlying operating system.

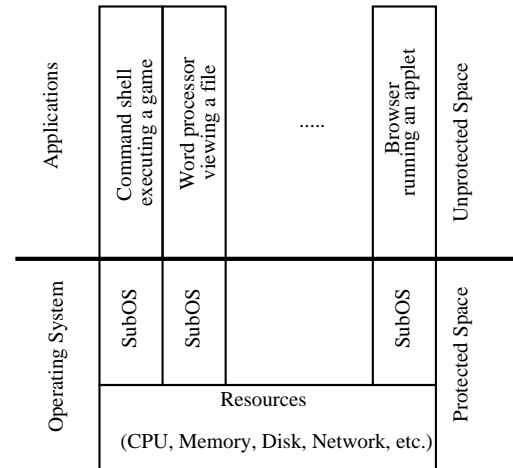


Figure 2. Under SubOS enabled operating systems user applications that “touch” possibly malicious objects no longer maintain the user access rights, and only get restricted access to the underlying system.

cuss work that is related to SubOS, and finally we conclude in Section 6.

2 Motivation

A number of trends in computing are fueling the need for a more flexible, yet stricter security model in operating systems.

2.1 Information Exchange

With the growth of the Internet, exchange of information over wide-area networks has become essential for both applications and users. Modern applications often fetch help files and other data over the World Wide Web. In extreme cases, like some versions of the BSD UNIX operating system, even whole operating systems install and upgrade themselves over the network. However, the most common case is electronic mail. Users regularly receive mail from unknown sources with a number of possibly malicious attachments. The attached documents use vulnerabilities in the helper applications that are invoked to process them, which in turn could compromise system security. The need for connectivity and exchange of information even at this most basic level is therefore a major threat to security.

It is also the case that seemingly inactive objects like Web pages or email messages are very much active and potentially dangerous. One example is JavaScript programs which are executed within the security context of the page with which they were down-loaded, and they have restricted access to other resources within the browser.

Security flaws exist in certain Web browsers that permit JavaScript programs to monitor a user’s browser activities beyond the security context of the page with which the program was downloaded (CERT Advisory CA:97.20). It is obvious that such behavior automatically compromises the user’s privacy.

Another example is the use of Multi-purpose Internet Mail Extensions (MIME). The MIME format permits email to include enhanced text, graphics, and audio in a standardized and inter-operable manner. Metamail(1) is a package that implements MIME. Using a configurable mailcap(4) file, metamail(1) determines how to treat blocks of electronic mail text based on the content as described by email headers. A condition exists in metamail(1) in which there is insufficient variable checking in some support scripts. By carefully crafting appropriate message headers, a sender can cause the receiver of the message to execute an arbitrary command if the receiver processes the message using the mailcap(4) package (CERT Advisory CA:97.14) [1].

2.2 Application Complexity

But the problem is deeper than obvious forms of mobile code. Given the increasingly complex environment presented to many applications, we assert that these applications have many of the characteristics of operating systems, and should be implemented as such.

Even simple HTTP requests return a complex object, wherein the remote side tells the local browser what to do,

up to and including a request to run certain applications. Print spoolers have to check file access permissions. Email can be delivered directly to programs. Web servers must run scripts, often via an interpreter, while denying direct access to the interpreter and perhaps ensuring that one script does not access or modify the private data of another script. All of these applications should worry about resource consumption. And these, of course, are the characteristics of operating systems. In fact, arbitrating access to various objects is more or less the definition of what an operating system does.

However, re-implementing an operating system with each new application would be extreme. Instead, our goal is to add sufficient functionality to an existing system so that applications can rely on the base operating system to carry out its own particular security policy. That security policy, in turn, can reflect its own particular needs and its degree of certainty as to the identity of users.

2.3 Inadequate Operating System Support

The lack of flexibility in modern operating systems is one of the main reasons security is compromised. The UNIX operating system, in particular, violates the principle of least privilege. The principle of least privilege states that a process should have access to the smallest number of objects necessary to accomplish a given task. UNIX only supports two privilege levels: “root” and “any user”.

To overcome this shortcoming, UNIX can grant temporary privileges, namely `setuid(2)` (set user id) and `setgid(2)` (set group id). These commands allow a program's user to gain the access rights of the program's owner. However, special care must be taken any time these primitives are used, and as experience has shown a lack of sufficient caution is often exploited [18].

Another technique used by UNIX is to change the apparent root of the file system using `chroot(2)`. This causes the root of a file system hierarchy visible to a process to be replaced by a subdirectory. One such application is the `ftpd(8)` daemon; it has full rights in a safe subdirectory, but it cannot access anything beyond that. This approach, however, is very limiting, and in the particular example commands such as `ls(1)` become unreachable and have to be replicated.

These mechanisms are inadequate to handle the complex security needs of today's applications. This forces a lot of access control and validity decisions to user-level software that runs with the full privileges of the invoking user. Applications such as mailers, Web browsers, word processors, *etc.*, become responsible for accepting requests, granting permissions and managing resources. All this is what is traditionally done by operating systems. These applications, because of their complexity as well as the lack of flexibility

in the underlying security mechanisms, possess a number of security holes. Examples of such problems are numerous, including macros in Microsoft Word, JavaScript, malicious Postscript and PDF documents, *etc.*

We wish to offer users flexible security mechanisms that restrict access to system resources to the absolute minimum necessary.

3 The SubOS Architecture

SubOS is a process-specific protection mechanism. Under SubOS any application (e.g. `ghostscript`, `Perl`, *etc.*) that operates on possibly malicious objects (e.g. Postscript files, `Perl` scripts, *etc.*) inherits the identity of that object. Any further system accesses that application makes are restricted by *that* identity, instead of the *user identity*. We will call these applications SubOS processes, or sub-processes in the rest of this paper. The access rights for that object are determined by a sub-user id that is assigned to it when it is first accepted by the system. The sub-user id is a similar notion to the regular UNIX user id's. In UNIX the user id determines what resources the *user* is allowed to have access to, in SubOS the sub-user id determines what resources the *object* is allowed to have access to. The advantage of using sub-user id's is that we can identify individual objects with an immutable tag, which allows us to bind a set of access rights to them. This allows for finer grain per-object access control, as opposed to per-user access control.

The idea becomes clear if we look at the example shown in Figure 3. Let us assume that our untrusted object is a postscript file `foo.ps`. To that object we have associated a sub-user id, as we will discuss in Section 3.1.1. `foo.ps` initially is an inactive object in the file system. While it remains inactive it poses no threat to the security of the system. However the moment `gs(1)` opens it, and starts executing its code, `foo.ps` becomes active, and automatically a possible danger to the system. To contain this threat, the applications that open untrusted objects, inherit the sub-user id of that objects, and are hereafter bound to the permissions and privileges dictated by that sub-user id.

There is a strong analogy here to the standard UNIX `setuid(2)` mechanism. When a suitably-marked file is executed, the process acquires the access rights of the owner. With SubOS, suitably-marked *processes* acquire the access rights of the owner of the *files* that they open. In this case, of course, the new rights are never greater than those the process had before.

The advantages of our approach become apparent if we consider the alternative methods of ensuring that a malicious object does not harm the system. Again using our postscript example we can execute `foo.ps` inside a safe interpreter that will limit its access to the underlying file system. There are however a number of examples on how

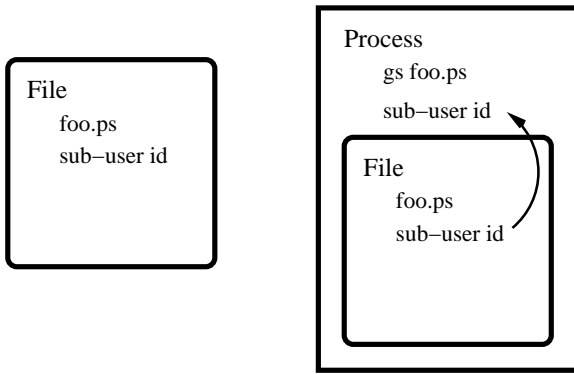


Figure 3. In the left part of the Figure we see an object, in this case a postscript file `foo.ps`, with its associated sub-user id. The moment the ghostscript application opens file `foo.ps`, it turns into a SubOS process and it inherits the sub-user id that was associated with the untrusted object. From now on, this process has the permissions and privileges associated with this sub-user id.

relying on safe languages fails [1]. We could execute the postscript interpreter inside a sandbox using `chroot(2)`, but this will prohibit it from accessing font files that it might need. Finally we could read the postscript code and make sure that it does not include any malicious commands, but this is impractical. Our method provides transparency to the user and increased security since every data object has its access rights bound to its identity, preventing it from harming the system.

3.1 Implementation

For our development platform we decided to use the OpenBSD operating system [2]. OpenBSD provides an attractive platform for developing security applications because of the well-integrated security features and libraries (an IPsec stack, SSL, KeyNote, *etc.*). However, there is nothing inherent in the SubOS architecture that limits us to UNIX like operating systems, so similar implementations are possible for operating systems like Microsoft Windows. The main advantage of our kernel implementation is that the additional security mechanisms will be largely transparent to the applications. Specifically, although the applications may need to be aware of the SubOS structure, they will not need to worry about access control or program containment.

3.1.1 Data Object Identifiers

As we mentioned earlier in Section 3, every time the system accepts an incoming object it associates a sub-user id with

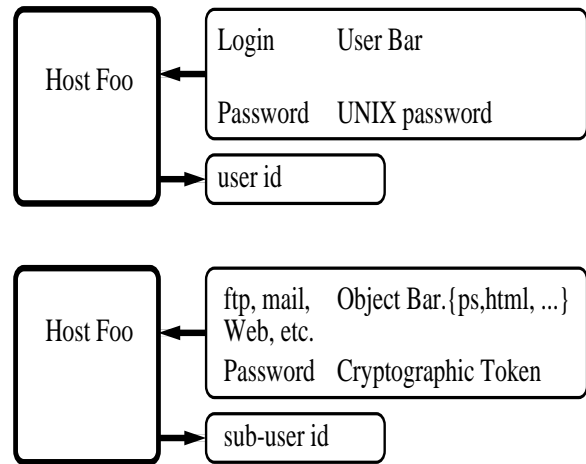


Figure 4. In the top part of the Figure we see the regular process of a user Bar logging in to a UNIX system Foo and getting a user id. In the same way objects that enter the system through ftp, mail, *etc.*, “log in” using a cryptographic token, and are assigned sub-user id’s.

it, depending on the credentials the object carries. The sub-user id is permanently saved in the Inode of the file that holds that object, which is now its immutable identity in the system and specifies what permissions it will have. It has essentially the same functionality as a UNIX user id. One can view this as the equivalent of a user logging in to the system.

Figure 4 shows the equivalence of the two mechanisms. In the top part of the figure we see the regular process of a user Bar logging in to a UNIX system Foo and getting a user id. In the same way, objects that enter the system through ftp, mail, *etc.*, “log in” and are assigned sub-user id’s based on their (often cryptographically-verified) source, as we will see in Sections 3.1.4 and 3.1.5.

Enhancing applications to utilize the functionality of the SubOS system require either making minor modification to the application code, or interposing a proxy that assigns sub-user id’s to objects arriving via proxied services, *e.g.* ftp, http, and mail.

3.1.2 Sandboxing

The most basic operation supported by SubOS is the inheritance of the sub-process id from an inactive file system object to a running process. To accomplish this we extended the `open(2)` system call. When it is used on objects that contain sub-user id’s, it copies the sub-user id to the `proc` structure of that process (Figure 3). At that point the process becomes a SubOS process bound to that sub-user id.

It is crucial that a sub-process can never “escape” its

sub-process status. To enforce this, whenever a sub-process forks and execs, the identity is inherited by the child process. To achieve this we extended the `fork(2)` and `exec(2)` system calls to have created processes inherit that status. Furthermore we modified the `creat(2)` system call, so that any files created by sub-processes have the sub-user id of the creator assigned in their Inode. Finally sub-processes are not allowed to execute setuid programs, to enforce this we block the setuid related (`setuid(2)`, `seteuid(2)`, `setgid(2)`, `setegid(2)`) system calls in the kernel.

It is not clear that that is the right choice. However, UNIX has traditionally had trouble when setuid programs invoked other setuid programs. To give just one historical example, in the days when the `mkdir(2)` call was implemented by executing a setuid—root program, subsystems that were themselves setuid had trouble creating directories.

3.1.3 Resource Protection

The SubOS mechanisms must protect the various resources of the users computer from viruses, Trojan Horses, worms, etc. In order to do so, it should monitor the creation of network connection, accesses to the file system, execution time of processes and allocation of physical memory, that might result from malicious code in untrusted objects.

By default a SubOS process is not allowed to create network connections. We accomplish this by filtering network related system calls. It is however possible to set up policies that will allow certain sub-processes to access the network by setting up the hosts they are allowed to connect to, the port, and the protocol to be used.

A practical implementation would require considerable attention to policies, including wild cards for port numbers, network masks for the host, etc. It might also be desirable to include certain known-safe local host/port combinations. For example, we may wish to permit open access to a local DNS proxy, for safe name resolution. On the other hand, wide-open access to a real name server might permit the controlled process to map local domains, which may be undesirable.

In order for the SubOS to restrict file system accesses we introduce the notion of a *view*. The view refers to the permissions a sub-process has to parts of the directory tree. Sub-processes don't use the permission bits that are normally used by processes (user, group, other). Rather, they have their own permissions that are defined in a configuration file, maintained by the user or administrator. This is very much like `chroot(2)` but more like pruning the directory tree of the file system than setting a new root.

The extended permission bits are added in lists in the inodes of the files specified in the configuration file. Every time the kernel identifies a file system access originating

from a sub-process, it traverses the list in the corresponding inode in order to locate the permissions that apply to the sub-user id of that sub-process. It then uses those permission bits, instead of the normal bits set for user, group or other, to determine whether to allow the access or not. If there are no permissions set for that sub-process the request is denied by default.

Finally execution time as well as memory allocation should also be monitored. This way, malicious objects (such as Java applets that run under a Web browser) will not hamper the smooth operation of the system. Our current working prototype lacks the appropriate controls for this type of enforcement, we are however in the process of implementing the necessary controls for the next version of our system. There are a number of things that need to be considered. Most importantly access to the `setpriority(2)` or `setrlimit(2)` system calls should be restricted, prohibiting sub-processes from executing at a higher priority than the parent process and limiting the amount of system resources they can allocate. Additionally, there must be a bound to the number of times a sub-process is allowed to *fork* in order to prevent possible process pollution in the system. Finally we need to add a form of accounting to monitor the amount of resources all the spawned sub-processes consume.

3.1.4 Sub-Users

In order for a SubOS to be effective, different sub-user ids must be assigned to different protection domains. Just how this is done depends on the application, on how the file arrived on the local system, and on any credentials it carries.

For emailed files, the sender's identity is used to select the sub-user id, naturally, such mail should be digitally signed. Mail from a previously-unknown user, or mail that cannot be assigned with enough confidence to a particular sender, receives a new sub-user id.

For Web browsers, finer-grained protection is desirable. Each site visited is assigned its own sub-user id, thus preventing one site from interfering with another's content. This could, for example, have prevented the "Frame Spoof" bug in Internet Explorer (MS98-020) [1].

3.1.5 Accessing Multiple Objects

So far we have assumed that sub-processes will operate on only one object at a time. However it is possible for a sub-process to open multiple objects, each with its own sub-user id. When a sub-process opens another object containing a sub-user id it also inherits that new id, and the new permissions would depend on a combination of the individual permission as dictated by the system policy.

This is easily accomplished in the case of CPU and memory allocation; the new sub-process will have the minimum

of the two for allocated memory and CPU time. In the case of network and file system access, any request is denied unless it is allowed by the permissions of *all* inherited sub-user id's.

4 SubOS Applications

To demonstrate the functionality of our SubOS enabled operating system we identified two applications that are most commonly targeted by hostile objects. Mailers and Web browsers are often attacked by a number of malicious attachments, and would therefore benefit from our security architecture.

4.1 A Secure Mailer

To test the functionality of our current prototype we modified a mailer, `mh(1)`, to take advantage of the SubOS architecture. To do this we extended `mh(1)` to implement a *login*-like mechanism. Depending on the source of the message—ideally, this should be cryptographically verified, though we have not yet implemented that portion—`mh(1)` will attach a sub-user id to that file when it saves it. `Mh(1)` assigns sub-user id's using a file, similar to the UNIX `/etc/passwd`, that matches e-mail addresses to id's.

Any helper application that is invoked—often automatically—to process the message, will inherit the sub-user identity. If the message contains active, malicious code, its effects will be contained by the limited permissions assigned to that sub-user id, protecting the rest of the system.

4.2 A Secure Browser

In our architecture we address the two security weaknesses of Web browsers:

- Helper applications running with the user's privileges.
- Web pages that carry active content that is interpreted by the browser.

To address the problems we will use the mechanism provided by the SubOS-capable operating system, as well as a modular Web browser architecture. The implementation is sketched here; a more complete description is in [16]. We divided the Web browser into three parts, according to its functionality. The first part is responsible for downloading objects over the network, the second is responsible for displaying the content, and the last is a set of helper applications/interpreters used to process the content of the downloaded objects. The design is presented in Figure 4.2

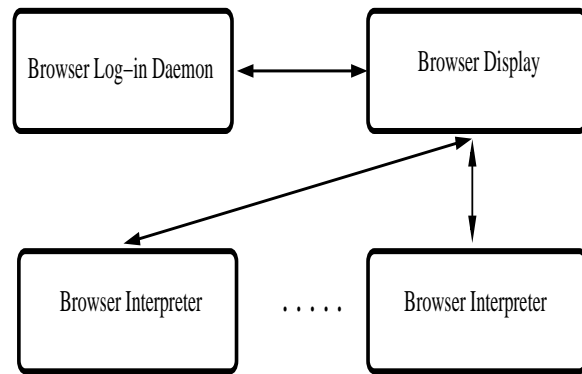


Figure 5. The Web browser is comprised of three parts. The first part is responsible for downloading objects from the net and assigning sub-user id's to them. The second provides the user interface of the browser. Finally the third is a set of processes that interprets the active code that is carried by the incoming objects.

4.2.1 Secure Browser Components

Every object that is downloaded by the browser log-in daemon is assigned a sub-user id, which is bound to some permissions, and is then stored in the file system, similarly to Section 4.1. Objects that carry certificates, such as pages downloaded from Web sites that use `https`, may be given more permissions than are unauthenticated objects. For example an authenticated object might get access to `/home/foobar`, network access and unlimited resources, whereas an unauthenticated object might only be granted access to `/tmp` with no access to the network and limited memory and cpu time allocation. The display process provides the user interface of the our Web browser. It can make requests to the log-in daemon to download files; it is also responsible for spawning interpreters to handle the incoming objects, and display HTML.

Any active code is executed within the context of the interpreters. They handle inline scripts like JavaScript, as well as other types of active code, such as Postscript and Perl. Since the objects they interpret are bound by their sub-user id, which was assigned to them when they first entered the system, they cannot cause any damage.

5 Related Work

The area of operating system security is a field that has received a great deal of attention, and has been researched extensively. However, the ever-increasing demand and need for communication and openness has put new strains on operating systems. Communication environments like the Internet require us to solve a whole new set of problems that

researchers have just recently started to address. In this section we focus our attention to work that is directly related to ours.

There are several methods for intrusion prevention in operating systems, ranging from type-safe languages [20, 22, 30, 14, 13], fault isolation [28] and code verification [25], to operating system-specific permission mechanisms [21, 26], system call interposition [12, 4, 3, 5] and system call interception [6, 10, 11, 7, 29, 24].

Capabilities and access control lists are the most common mechanisms operating systems use for access control. Such mechanisms expand the UNIX security model and are implemented in several popular operating systems, such as Solaris and Windows NT [8, 9]. However they offer no protection for the user against programs owned by the user, which may contain errors, Trojan Horses, or viruses.

The Flask system [26] extends the idea of capabilities and access control lists by the more generic notion of the *security policy*. The Flask micro kernel system relies on a security server for policy decisions and on an object server for enforcement. Every object in the system has an associated security identifier very similar to our notion of a sub-user id. Requests coming from objects are bound by the permissions associated with their security identifier. However Flask does not address the threat SubOS is trying to protect against, namely passive objects becoming active and then executing with the permissions of the running process. As a minor issue, we have demonstrated that our prototype can be easily implemented as part of a widely used, commodity operating system, as opposed to an experimental micro kernel.

The traditional Orange Book-style systems offer protection against violation of security levels by malicious programs. But there is no barrier to attacks on files at the current security level, nor to attacks at that security level over the network. For example, a Top Secret worm can still be able to spread, though it would only be able to infect other Top Secret-rated systems.

Reeds and McIlroy's unique implementation of the Orange Book's security policies [23] bears a strong conceptual resemblance to the SubOS scheme. Rather than assigning a process or a file fixed access rights or labels, these "float" in response to the program's execution. A process that opens a file marked Top Secret acquires a Top Secret label; any files that it writes are also marked Top Secret. Permissions are thus data-driven, as in SubOS.

A different approach relies on the notion of call interposition. Systems like [12, 4, 3, 5] operate at user level and confine applications by filtering access to system calls. To accomplish this they rely on `ptrace(2)`, the `/proc` file system, and special shared libraries. Another category of systems [6, 10, 11, 7, 29, 24], goes a step further. They intercept system calls inside the kernel, and use policy en-

gines to decide whether to permit the call or not. Our system differs in a major point. We view every object as a separate user, each with its own sub-user id and access rights to the system resources. This sub-user id is attached to every incoming object when it is accepted by the system, and stays with it throughout its life, making it impossible for malicious objects to escape.

In [17] the authors identify the dangers of active content and the need to contain it. They authenticate incoming objects and grant them access rights. These access rights identify which interpreters are allowed to operate on the objects. Furthermore these interpreters are also "sanitized" so that they don't include any unsafe calls. Our system offers much finer access control, enforced by the operating system kernel.

The methods that we mentioned so far rely on the operating system to provide with some sort of mechanism to enforce security. There are, however, approaches that rely on safe languages, [20, 27, 19, 15] the most common example being Java [22, 13]. In Java applets, all accesses to unsafe operations must be approved by the security manager. The default restrictions prevent accesses to the disk and network connections to computers other than the server the applet was down-loaded from. Our system is not only restricted to a limited set of type safe languages. We can secure any process running on the system that has touched some untrusted object.

Code verification is another technique for ensuring security. This approach uses *proof-carrying code* [25] to demonstrate the security properties of the object. This means that the object needs to carry with it a formal proof of its properties; this proof can be used by the system that accepts it to ensure that it is not malicious. Code verification is very limiting since it is hard to create such proofs. Furthermore, it does not scale well; imagine creating a formal proof for every Web page.

6 Conclusions

We have designed and implemented an object-specific protection mechanism to contain untrusted data. We restrict the environment that such objects can operate in, and the resources they can access, by extending the UNIX security model to assign sub-user id's to them and then treating them like regular users. The implementation is part of the kernel of the operating system, since that is the only natural and secure place for security mechanisms to enforce policies. SubOS is a working prototype implemented as part of the OpenBSD operating system. Finally, we have shown how SubOS relates to other security mechanisms and how it strengthens operating system security.

Acknowledgements

This work was supported by AT&T, by the DoD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795, and DARPA under Contracts F39502-99-1-0512-MOD P0001 and F30602-01-2-0537.

References

- [1] CERT Advisories. <http://www.cert.org/advisories/>.
- [2] The OpenBSD Operating System. <http://www.openbsd.org/>.
- [3] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. In *Proceedings of the 2000 USENIX Security Symposium*, pages 1–17, Denver, CO, August 2000.
- [4] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations, December 1998.
- [5] R. Balzer and N. Goldman. Mediating connectors: A non-bypassable process wrapping technology. In *Proceeding of the 19th IEEE International Conference on Distributed Computing Systems*, June 1999.
- [6] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of the USENIX 1995 Technical Conference*, New Orleans, Louisiana, January 1995.
- [7] C. Cowan, S. Beattie, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Security for Server Appliances. In *Proceedings of the 14th USENIX System Administration Conference (LISA 2000)*, Mar. 2000.
- [8] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.
- [9] H. Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.
- [10] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.
- [11] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 39–52, June 1998.
- [12] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 1996 USENIX Annual Technical Conference*, 1996.
- [13] L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.
- [14] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, 1996.
- [15] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Programming Language for Active Networks. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, February 1998.
- [16] S. Ioannidis and S. M. Bellovin. Building a Secure Browser. In *Proceedings of the Annual USENIX Technical Conference, Freenix Track*, June 2001.
- [17] T. Jaeger, A. D. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of the 1996 USENIX Security Symposium*, pages 131–148, San Jose, Ca., 1996.
- [18] R. Kaplan. SUID and SGID Based Attacks on UNIX: a Look at One Form of the Use and Abuse of Privileges. *Computer Security Journal*, 9(1):73–7, 1993.
- [19] X. Leroy. Le système Caml Special Light: modules et compilation efficace en Caml. Research report 2721, INRIA, November 1995.
- [20] J. Y. Levy, L. Demailly, J. K. Ousterhout, and B. B. Welch. The Safe-Tcl Security Model. In *USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 1998.
- [21] D. Mazieres and M. F. Kaashoek. Secure Applications Need Flexible Operating Systems. In *The 6th Workshop on Hot Topics in Operating Systems*, May 1997.
- [22] G. McGraw and E. W. Felten. *Java Security: hostile applets, holes and antidotes*. Wiley, New York, NY, 1997.
- [23] M. D. McIlroy and J. A. Reeds. Multilevel security in the unix tradition. *Software Practice and Experience*, 22(8):673–694, 1992.
- [24] T. Mitchem, R. Lu, and R. O'Brien. Using Kernel Hypervisors to Secure Applications. In *Proceedings of the Annual Computer Security Applications Conference*, Dec. 1997.
- [25] G. C. Necula and P. Lee. Safe, Untrusted Agents using Proof-Carrying Code. In *Lecture Notes in Computer Science, Special Issue on Mobile Agents*, October 1997.
- [26] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, and J. Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the 2000 USENIX Security Symposium*, pages 123–139, Denver, CO, August 2000.
- [27] J. Tardo and L. Valente. Mobile Agent Security and Telescript. In *Proceedings of the 41st IEEE Computer Society Conference (COMPCON)*, pages 58–63, February 1996.
- [28] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.
- [29] K. M. Walker, D. F. Stern, L. Badger, K. A. Oosendorp, M. J. Petkac, and D. L. Sherman. Confining root programs with domain and type enforcement. In *Proceedings of the 1996 USENIX Security Symposium*, pages 21–36, July 1996.
- [30] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

Session 7. Peer-to-Peer

1. The Design of a Robust Peer-to-Peer System
Authors: Rodrigo Rodrigues, Barbara Liskov, Liuba Shrira,
page 116
2. Self-Organization in Peer-to-Peer Systems Jonathan Ledlie
Authors: Jacob Taylor, Laura Serban, Margo Seltzer
page 125
3. HiScamp: self-organizing hierarchical membership protocol Ayalvadi J. Ganesh, Anne-Marie Kermarrec, Laurent Massoulie Authors:
page 133
4. One Ring To Rule Them All: Service discovery and binding with a distributed hash table Miguel Castro, Peter Druschel, Antony Authors: Rowstron
page 140

The Design of a Robust Peer-to-Peer System

Rodrigo Rodrigues, Barbara Liskov, Liuba Shrira*
MIT Laboratory for Computer Science
{rodrigo, liskov, liuba}@lcs.mit.edu

Abstract

Peer-to-peer (P2P) overlay networks have recently become one of the hottest topics in OS research. These networks bring with them the promise of harnessing idle storage and network resources from client machines that voluntarily join the system; self-configuration and automatic load balancing; censorship resistance; and extremely good scalability due to the use of symmetric algorithms. However, the use of unreliable client machines leads to two defects of these systems that precludes their use in a number of applications: storage is inherently unreliable, and lookup algorithms have long latencies. In this paper we propose a design of a robust peer-to-peer storage service, composed not of client nodes, but server nodes that are dedicated to running the peer-to-peer application. We argue that our system overcomes the defects of peer-to-peer systems while retaining their nice properties with the exception of utilizing spare resources of client machines. Our system is capable of surviving arbitrary failures of its nodes (Byzantine faults) and we expect it to perform and scale well, even in a wide-area network.

1 Introduction

We have witnessed the recent emergence of a number of peer-to-peer (P2P) distributed systems, i.e., systems in which all nodes have identical responsibilities and all communication is symmetric. Successful applications of these systems include content sharing [7] and large-scale storage systems [6].

Peer-to-peer computing offers several advantages over other traditional distributed systems, such as automatic load balancing and self-organization. But perhaps the most valuable feature of these systems is that, due to the symmet-

ric nature of peer-to-peer, the desirable properties of the system can scale when new nodes are added to the system. These properties include (but are not limited to) performance, availability, and fault-tolerance.

Part of the success of these systems comes from their ability to harness idle storage and network resources, offered by everyone who is willing to participate in the system. Unfortunately, such resources are inherently unreliable: we cannot control what code is running in these machines, and they will join and leave the network frequently. This leads to two essential defects of P2P storage systems: they offer weak guarantees in terms of storage reliability; and they must include complicated lookup algorithms (with very high latency) to allow low cost routing information updates in the presence of frequent joins and leaves.

In this paper we describe the design of a P2P storage system that solves these problems. Our system provides reliable storage in the presence of failstop failures. In addition it also performs reliably in the presence of Byzantine failures: the storage service will continue to work correctly even if faulty components behave arbitrarily. Our system overcomes the main problems of traditional P2P storage systems while maintaining their essential design philosophy and the advantages that derive from it.

Our system has a hybrid architecture, consisting of a set of servers — not unreliable client machines that participate in the system intermittently — acting as the P2P nodes, and a *configuration service* (CS). The CS runs on a set of special servers (presumably less failure-prone than P2P nodes). It is responsible for computing the current configuration (including removing faulty P2P nodes from the system), and informing the P2P nodes about the current configuration.

Not all applications require the kind of robustness this system provides. In particular, applications like the Cooperative File System (CFS) [3] assume that the file system is created in a secure place and the untrusted peer-to-peer storage is used only to publish information. Therefore, if the peer-to-peer system loses state, the information can be refreshed from the original source. Our techniques are required, however, for applications that store their actual state, rather than a copy of the state, in the P2P system. Examples

* Department of Computer Science, Brandeis University

This research was supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory. Rodrigo Rodrigues is supported by a Praxis XXI fellowship.

of such applications include various kinds of archival services, ranging from digital libraries to archives for file systems and object-oriented systems; mail services; key distribution services; and databases that store large amounts of information such as astronomical data.

The next section discusses P2P computing and motivates our approach by explaining why P2P systems as presently designed cannot tolerate Byzantine-faulty nodes and what is needed to fix the problem. Section 3 presents our system model and lists our assumptions. The configuration service is presented in Section 4 and the P2P nodes in Section 5. We conclude in Section 6.

2 Motivation

This section presents a brief description of P2P systems and then explains why these systems cannot tolerate Byzantine failures. Then it describes what is necessary to solve the problems: P2P servers that do proactive recovery and a configuration service that determines the current configuration using Byzantine agreement protocols and propagates the information to the P2P nodes.

2.1 Peer-to-Peer Computing

Recently proposed P2P storage systems provide the abstraction of a distributed hash table [12, 14, 18, 19]. Each data item is assigned an ID, and the system provides the ability to store the item with that ID and to retrieve it later. Applications choose the IDs and this is typically done in a way that allows the data to be self-verifying.

The system is composed of nodes each of which is assigned an ID. The system stores an item at a node or nodes based on a computation that takes into account the item ID and the node IDs.

The functionality of the P2P nodes is divided into a lookup layer that locates the nodes responsible for an item given its ID, and a storage layer built on top of the lookup layer that provides the distributed hash table abstraction. Each of these layers provides both client and server functionalities: the client storage layer uses the client lookup layer to locate servers that hold the desired data. The server storage layer is responsible for storing the data items, maintaining proper levels of replication, and performing caching. The server storage layer and lookup layer interact in order to maintain the correct mapping between the servers that are present in the system and the data they hold.

Node IDs are typically computed by applying a collision-resistant base hash function such as SHA-1 [16] to information about the node, such as its IP address, resulting in an m -bit identifier. Data item IDs are also of the same length (and are often computed by hashing the content of the item).

Different peer-to-peer lookup systems use different criteria to determine which node is responsible for storing a particular data item. For instance, Chord [18] is based on consistent hashing [8], where identifiers are ordered in an identifier circle modulo 2^m , and the data item with ID i is assigned to the first node whose identifier is equal to or follows i in the identifier space (called the *successor node* of ID i). Pastry [14], on the other hand, allows applications to store data items in the nodes with IDs numerically closest to the ID of item being stored. Without loss of generality, we will assume the use of consistent hashing in our system description. Changing the system to use other mappings should be straightforward.

The storage layer builds a reliable storage service on top of the lookup layer. Reliability cannot be achieved without replication, and therefore this layer must make decisions on where to store the replicas of the data. Typically, there is a great deal of interdependence between the replica placement decisions and the details of the lookup algorithms. For instance, the DHash layer built on top of Chord [3] stores the data item not only at the successor of its ID i but also at the next n successors. These are easy to determine since the Chord layer maintains a list of successors for each node. Similarly, PAST [15] stores replicas on the n nodes whose ID are numerically closest to the item ID i , which is also information contained in the data structures maintained by the Pastry lookup algorithm.

Our system can easily adapt to any placement strategy. We will assume, without loss of generality, that we store each data item at the n successors of ID i in the ID space (similarly to DHash). We believe that changing this placement strategy does not affect the properties of our system, as long as the strategy is based on randomly-chosen node IDs. This is important so that the likelihood of replicas of each data item failing is as independent as possible. Random node IDs avoids, for instance, having all replicas in the same LAN, which would make a simple disconnection of the LAN from the Internet sufficient to make the data unavailable.

2.2 Problems in Peer-to-Peer Systems

Most currently proposed peer-to-peer systems are designed to tolerate failstop failures (e.g., [14, 18]). They assume replicas fail by stopping or omitting some steps. Two aspects of these algorithms make them particularly vulnerable to malicious attacks.

First, there is no admission control. This implies that an attacker can join the system with multiple personalities that offer a large amount of resources (even if the attacker does not own such resources, as described in [5]) and this way the attacker can control a substantial fraction of the nodes and increase the probability of a successful attack.

Second, all nodes trust the configuration information they hear from other nodes to be correct. If this information is incorrect (because it comes from a Byzantine-faulty node), this can cause a client request to be diverted to a parallel, internally consistent system formed only of malicious nodes that pretend to store the data correctly but can choose to not allow correct retrieval [17]. Or even more seriously, a P2P node might update its routing state incorrectly, to reflect a bogus configuration change reported by a malicious node.

Solving this second problem is difficult, and its solution motivates the main architectural choices of our system: we need a way to reliably detect faulty nodes and to remove them from the system. As we will explain, the configuration service provides this ability.

2.2.1 Detecting Failed Nodes

Knowing a node is faulty is difficult even if we are only trying to detect failstop failures. Typically, fault detection can be done by some sort of ping protocol where you periodically test that the node is alive and reachable. Assuming the node is faulty when you cannot ping it is not reasonable in the presence of denial of service attacks. Still, if we assume denial of service attacks are bounded we can reason this way.

Detecting Byzantine faults that result, for instance, from a malicious attack is much harder. This is so because nodes can appear to behave properly even if they have been compromised, and therefore obtaining a proper reply from a node does *not* imply it is correct. Also, it is impossible to check if a node is correct by inspecting its state, since a faulty node could appear correct when inspected and misbehave when replying to clients. Thus, the only way to detect Byzantine faults by probing is if the probes are indistinguishable from client requests. But even with such probes, we can't guarantee to detect Byzantine-faulty nodes in time, because an attacker can compromise more and more nodes, making them behave properly until enough have been compromised to cause the system to malfunction. In other words, it is impossible to get rid of Byzantine faulty nodes in a timely way using probes.

Thus instead of probing, we rely on a proactive recovery mechanism [2] to make Byzantine-faulty nodes behave correctly again. In this scheme, all nodes in the system are recovered proactively and frequently, even if there is no reason to suspect they are faulty. When a node is recovered, it is rebooted and restarted from a read-only disk that contains a correct version of the code. Then it is brought up-to-date by fetching its missing or incorrect parts of the service state from nodes that contain copies of that state. This mechanism allows us to assume that nodes are Byzantine faulty only for short periods, so that we do not need to worry about

detecting and removing Byzantine faulty nodes. Thus we will concentrate on removing unreachable nodes from the system.

Note that proactive recovery makes it unlikely that P2P nodes could be client machines that voluntarily join the system and provide their resources to it. Instead they must be server machines dedicated to handling the distributed application. However, as discussed earlier, there are other reasons why we want to use servers as the P2P nodes. Server machines are less failure-prone than client machines (which minimizes the probability of enough machines being compromised so that the system becomes unavailable), and they are not constantly joining and leaving the system, which facilitates management of the routing state. However, there can still be huge numbers of nodes in the system, provided by many different organizations; they can still run symmetric protocols; and the nodes can still be distributed across a wide area, allowing them to survive catastrophic failures (from earthquakes to terrorist attacks).

2.2.2 Removing Faulty Nodes

We still need to figure out how to decide what nodes should be removed from the system and we also need to propagate this information to the P2P nodes in a way that cannot be subverted by Byzantine-faulty nodes.

Deciding which nodes are faulty requires some form of agreement, since we cannot trust an individual node to be correct and make the right decision. Therefore the decision must be made by a group of replicas that carry out an agreement protocol that is robust in the presence of Byzantine failures. These replicas must run a Byzantine agreement protocol [1, 2] to agree on the correct state of the configuration.

Our architecture uses the configuration service (CS) for this purpose. The CS controls membership in the current P2P configuration and periodically notifies the P2P nodes of configuration changes. It uses an agreement protocol to decide on changes, and it propagates configuration information to the P2P nodes, authenticated using digital signatures so that the P2P nodes can be certain that the information is correct.

The CS could run on a subset of the P2P nodes. It could only run on a subset, rather than on all the P2P nodes, because the CS replicas need to communicate with all other nodes, and they need to carry out agreement protocols, so this would be impractical if all P2P nodes were involved. The subset might be selected statically, but that would go against our self-configuring design principle. Or we could imagine that it is selected dynamically: part of defining the next configuration is choosing the subset of nodes that will be the CS for that configuration.

However there are advantages to keeping the CS separate

from the P2P nodes. The CS nodes can be more reliable than the P2P nodes, with hardened security (e.g., physically isolated, geographically diverse, running different software and with different administrators). In addition the CS nodes have lots of work of their own to do; having P2P nodes do this work might lead to excessive load.

3 System Model

We assume an asynchronous distributed system where nodes are connected by a network. The network may fail to deliver messages, delay them, duplicate them, or deliver them out of order. We use a Byzantine failure model, i.e., faulty nodes may behave arbitrarily.

To authenticate communication in the presence of Byzantine faults, we rely on cryptographic techniques that an adversary cannot subvert. Not only do we need such protocols for communication within the CS and from the CS to the P2P nodes, we also require them occasionally for communication between P2P nodes (as discussed below). Therefore we assume that each node (both CS and P2P) has a secure cryptographic co-processor (which prevents exposure of a node's private key), a read-only disk where it stores the correct service code, and a watchdog timer that triggers recoveries. These are common assumptions for Byzantine fault tolerance algorithms [2].

We assume that all nodes in the system initially get to know the identity and public keys of the replicas in the CS using an out-of-band mechanism. When admitting a P2P node in the system, the CS gets to know its public key as well. This information is propagated to the P2P nodes as part of the configuration information.

We allow for a very strong adversary that can coordinate faulty nodes, delay communication, or delay correct nodes in order to cause the most damage to the replicated service. We do assume that the adversary cannot delay correct nodes indefinitely, and cannot cause an arbitrary delay to messages that are sent to reachable nodes (i.e., there are bounds on the duration of a denial-of-service attack).

4 The Configuration Service

The CS is responsible for determining the current configuration of the system, and propagating this information to the P2P nodes, so that they know what other nodes to contact to store or retrieve data. The CS replicas carry out a Byzantine-fault-tolerant protocol [1] to ensure that they agree about configuration state; this is necessary to ensure that the configuration state is correct and consistent.

This service performs four main functions that are described in the next sections.

4.1 Admission Control

The CS controls nodes entering the system, since otherwise, as discussed earlier, a malicious party can subvert the system.

The simplest way to do admission control is for the CS to maintain a list of authorities who are permitted to add nodes to the system. Each request to add a node must be signed by one of these authorities.

In addition, the CS provides a way to add and remove authorities.

When a node joins the system, the CS must be informed about its ID, its IP address, and its public key. This information will be propagated to the P2P nodes in the next configuration description.

4.2 Node Monitoring

The CS monitors the availability and reachability of the nodes using a ping protocol. Each CS replica must do its own monitoring of all the P2P nodes. This is needed so that it can form its own view of which nodes are faulty; then it will be able to decide whether a configuration change proposed by some other replica is reasonable.

CS replicas can do monitoring by sending pings to the P2P nodes. Alternatively, pings could be initiated by the P2P nodes. A good time to do this is when the P2P node restarts after proactive recovery since at that moment, the node is up and not faulty. We plan to use a combination of these techniques since each has its advantages. If pings are initiated by the P2P nodes there is less traffic since pings need not be acknowledged. Initiating the pings by the CS allows us to control the periodicity of the pings. This way we can increase the frequency of pings when we suspect a node is down, so that we can verify this more credibly.

The results of the pings are inserted in a *liveness database* local to the CS replica.

4.3 Deciding on a New Configuration

Periodically the CS nodes must decide on a new configuration. The new configuration will contain all the new nodes that joined the system since the last configuration was produced. The new configuration will *not* contain nodes that the CS replicas agree are faulty.

CS nodes scan the liveness database and try to evict potentially faulty nodes from the system. Having the CS nodes agree on evicting someone is not trivial, as different nodes will have different values for how long each node has been unreachable. We solve this using the non-deterministic choices validation mechanism proposed in [13]. CS nodes initiate a node eviction operation on the CS group if they haven't heard from a node for longer than $T + \epsilon$ time units.

The decision to evict a node is a non-deterministic choice, and therefore it needs to be validated by other CS replicas. They should accept the operation if they haven't heard from the same node for longer than T time units. This approach ensures that most eviction operations will succeed.

The eviction threshold $T + \epsilon$ is chosen to be longer than the assumed bound on denial-of-service attacks. It should be enough longer that the probability of evicting a non-faulty node because the attack prevents communication with it is small.

4.4 Propagating Configuration Information

The CS produces new configurations periodically (e.g., once every hour). Doing this only once in a while makes the system much more practical. We require that new configurations be generated often enough to preserve the following correctness condition for the P2P nodes:

At any moment, for any group of $n = 3f + 1$ replicas of a data item, that group contains no more than f faulty replicas.

A configuration description includes start and expiration times. The new configuration has a start time equal to the expiration time of the previous configuration. We assume that all participants have loosely synchronized clocks that allow them to perceive similar configuration intervals. The assumption about loosely synchronized clocks is a reasonable one for current systems due to the use of clock synchronization protocols [11]. (If these protocols do not provide adequate level of fault-tolerance, GPS can be used instead.)

New configurations must be signed by the CS and propagated to all the nodes, e.g., using gossip methods [4] to avoid overloading the CS. Signing configurations is not trivial, since an attacker can compromise one replica of the CS at a time, and have each replica sign wrong future configurations that are later combined to form a valid signature. We could solve this by having all P2P nodes read the configuration state from the CS nodes (from $2f + 1$ of them) periodically, but this is a heavy weight solution that precludes the use of gossip methods and increases load on the CS. One possible solution is to employ threshold cryptography methods for signing certificates as proposed in [20].

We have choices about what configuration information to propagate to which nodes. For example, we could partition the configuration information and each P2P node would learn only the information needed for it to carry out its base algorithm. Thus in a Chord system [18], nodes would be given their successors and their fingers.

But having to deal with incomplete configuration information is a problem in P2P systems. Traditional peer-to-peer systems have to cope with nodes frequently joining and leaving the network, and therefore try to limit the amount of routing state that has to be updated when configuration

changes occur. The penalty for this is having to use lookup algorithms with high latency: each lookup involves contacting a substantial number — typically $O(\log(N))$ — of nodes [12, 14, 18, 19]. In our system, however, we assume that nodes join and leave the system much less frequently than client machines in traditional P2P systems, and these node addition and removal operations are grouped together in even less frequent configuration changes dictated by the CS.

Therefore it seems reasonable to take advantage of the CS to simplify the routing algorithm by disseminating the entire configuration information to all the P2P nodes. Thus, routing decisions can be made locally, avoiding the latency of contacting a series of nodes.

The main cost of this approach is that the P2P nodes must store all this information. But actually the amount of storage required is small, even when we scale to hundreds of thousands of nodes. If we assume that the configuration consists, for each node in the system, of a 160 bit node identifier (based on a SHA-1 cryptographic hash function), plus its IP address, port number, and 1024 bit RSA public key, then the entire configuration will fit in approximately 14.7 megabytes, when we scale to 100,000 nodes. This information can easily fit in main memory.

Transmitting the information from the CS to the P2P nodes is not an issue since this can be done using diffs. The configuration description must contain a signed certificate containing the start and end time of the configuration and a fingerprint of its current state, so that nodes can communicate about configurations reliably without having to send the entire configuration.

In addition, the CS needs to allow P2P nodes to read the entire configuration; this is necessary when a node first starts using the system, or if it becomes very out of date. This is a large amount of information to be transmitted, but some of it is highly compressible (e.g., IP addresses and port numbers). Also, we could devise a mechanism where only the current configuration and specific public keys are obtained immediately, and the remaining public keys are obtained in the background. After a node is temporarily disconnected we can minimize the amount of information transmitted by using Merkle trees [10] to determine exactly the subset of the configuration that is out-of-date.

5 Peer-to-Peer Nodes

This section describes the processing at the P2P nodes. We designed our system with an application like CFS in mind, but extended the model to provide robustness. In the future, we intend to investigate what aspects of this design are not suitable for other applications and refine it to overcome these problems.

Figure 1 illustrates the software structure of the P2P

nodes. The figure makes a distinction between clients and servers. However, a server machine can also act as a client, similar to what happens in other P2P systems; in this case the application layer must be prepared for the unavailability of the node during proactive recovery. Alternatively, there may be dedicated clients that do not implement server functionality.

Other than the possibility of dedicated clients, the software structure is similar to what is used in previous P2P systems. In the description of the P2P nodes, we will use the term “client” to refer to the client functionality of the P2P nodes, which may correspond either to the client subset of the P2P node functionality, or to a dedicated client.

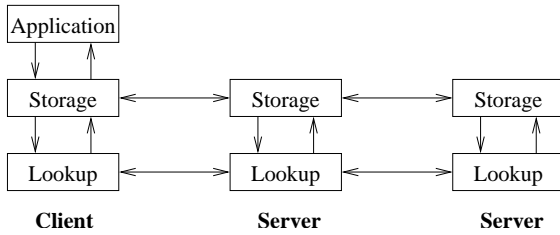


Figure 1. Software Structure

5.1 Lookup Layer

Periodically, the lookup layer receives information from the CS about the latest configuration. The lookup layer will notify the storage layer of this fact. In the new configuration, it is likely that the set of P2P nodes storing certain data items will change; the storage layer must do state transfer (described in Section 5.2.2) to move those items to the nodes that now store them.

The lookup layer on the client side also maintains information about the latest configuration. This enables clients to go directly to the P2P nodes that hold the data of interest: the storage layer at the client side asks the lookup layer for the replicas that hold the data item with a particular ID, and then it can contact the storage layer at the replicas directly.

P2P nodes obtain the configuration information from the CS the first time they join the system, but after this they can obtain new configurations directly from other nodes.

5.2 Storage Layer

Our storage layer must make a distinction between two types of data that can be stored in the system:

1. Immutable, self-verifying data. For instance, CFS data blocks do not change over time and are indexed by the hash of its contents, and therefore their integrity can be verified.

2. Mutable mappings that may be subject to a replay attack; CFS root blocks are an example. For such data we need to make sure we are retrieving the latest version. We do expect this data to be signed so that its integrity can also be verified by clients.

To ensure freshness of the second type of data, the client must extend it with a monotonically increasing version number that is also covered by the signature.

We will now present the algorithms that are involved in the operations of the storage layer. We will begin by describing how we store and retrieve data when the configuration does not change, and then describe what happens during configuration changes — how state transfer takes place and how the storage and retrieval algorithms work in the presence of a changing configuration.

5.2.1 Storage and Retrieval with a Static Configuration

In general, the storage algorithms for both types of data must ensure that $2f + 1$ replicas claim to have stored the data. The reason why we do not wait for the remaining f replicas is that they may be faulty and are never going to respond. Therefore, we need to make progress after hearing from only $n - f = 2f + 1$ replicas.

Thus for a client to write data of either type, it must send the write to at least $2f + 1$ replicas. It needs to receive acknowledgments from $2f + 1$ replicas before it can be certain that the write succeeded.

In all storage operations, the integrity of the data should be verified by the replicas storing that data item to avoid garbage being stored. For data of type (1), this means verifying that the ID is a hash of the content.

Reading data of type (1) is simple: it is sufficient to read from one replica, as long as the client verifies that the hash of the contents matches the ID. If it does not, the client repeats the read but this time asks for all the replicas of the data until it finds one that matches.

Version numbers for writes of data of type (2) are generated by the clients, and storage nodes reject blocks with version numbers not greater than the current version that is stored. The client doing the write can simply generate this version number (through prior knowledge or storage outside the system). Or, it can read the current version to learn the current version number; then it can increment that number to obtain the new version number and use it when writing the data.

Reads for data of type (2) have to wait for $2f + 1$ replies and choose the block with the highest version number. Note that reading from $2f + 1$ replicas has the nice property that the set of replicas we read from intersects the set of replicas for the last write in at least one non-faulty replica. This can

be seen as a particular instance of a dissemination Byzantine quorum system [9].

When reading data of type (2), clients must ensure they are receiving data from the correct nodes, since otherwise they could be tricked into a replay attack where they would be presented out-of-date versions of the data. Therefore, requests for data of this type contain a nonce that is sent with the signed reply to ensure its authenticity and freshness. As mentioned, clients know the public keys of P2P nodes since this is part of the configuration information.

One problematic situation is a faulty client sending different data items of type (2) to be stored at different replicas with the same version number. We address this issue by allowing temporary inconsistencies (different reads may obtain different values for the data item in question after the faulty write) which will be solved when a non-faulty client overwrites that data item. This temporary inconsistency might be avoided by having the replicas perform a consensus on the value being stored, but we decided against this because of the slowdown it imposes, especially in a wide area network.

These algorithms assume, like CFS does, that there are no concurrent writes to the same data item. Serializability for concurrent data accesses can be implemented by ordering the writes by version number and, if the version numbers are the same, by the hash of the contents.

5.2.2 State Transfer

When the configuration changes, the P2P nodes that are responsible for storing a particular data item in the new configuration may differ from those storing it in the previous configuration. In this case, state transfer must take place to bring the new replicas for that data item up-to-date.

This task can be simplified if all replicas involved in this process have access to both the old and the new configurations. This way everyone knows which nodes contain data they need, and which nodes need data they have.

So assume that both configurations are distributed to all nodes (at different times, though). When a node receives a new configuration ($n + 1$), and realizes it is now responsible for some new data items, it reads that data from the nodes that held it in configuration n . Nodes receiving data from configuration n should check the validity of the sender, the integrity of the data, and, if it is of type (2), should merge the copies it receives from different replicas (i.e., keep the one with the latest version number). Note that for data of type (1) it is enough to receive the data from one replica, whereas for data of type (2) the node must receive data from $2f + 1$ replicas and pick the one with the highest version number.

We also assume that nodes include identifiers of the latest configuration they know in the messages they exchange.

If a node detects an identifier greater than its current configuration, it asks the node that sent the identifier for a copy of that configuration. The same applies to clients: if a client tries to store a data item and a storage node that it talks to knows a configuration later than the one the client is using, it forces the client to fetch the new configuration, and the client repeats the store operation in the new configuration. This prevents different clients from performing concurrent reads and writes in different configurations, which could lead to inconsistencies.

6 Conclusions

In this paper we presented techniques to build Byzantine fault-tolerant peer-to-peer systems. To achieve this level of robustness in peer-to-peer systems we needed to deviate from traditional peer-to-peer architectures in two fundamental ways. First, we proposed a hybrid system, with symmetric storage nodes that implement the peer-to-peer system and a configuration service that performs admission control and determines the current configuration. Second, the P2P nodes (and also the CS nodes) must be server machines dedicated to run the storage application, and not client machines that run arbitrary code and are constantly entering and leaving the system. Our server machines have secure co-processors and perform proactive recovery.

We also sketched the design of a system built according to these principles. We believe that this design will work and the resulting system should perform well. The P2P nodes will not be burdened with determining membership and therefore can perform their storage function without interference except during configuration changes, which occur infrequently. In addition, because the P2P nodes and the clients store the entire configuration, routing is very efficient.

There is much future work that needs to be done, however. There are still many design issues to be resolved, and then we need to implement the system. We also plan to study the use of the system in various applications, and to extend it as needed. For example, one needed extension is providing support for a delete operation and a quota system to limit the amount of storage a client can use. The two go together, because when a quota system is deployed, a client trying to write to the system when its quota is reached needs to have some way of freeing up space to be able to continue using the system. We are working on algorithms to support these two features.

Acknowledgements

We would like to thank Emil Sit, Steven Richman, Chuang-Hue Moh, Sameer Ajmani, and the anonymous referees for their helpful comments on drafts of this paper.

References

- [1] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI-99)*, pages 173–186, New Orleans, Louisiana, February 1999.
- [2] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 273–288, San Diego, California, October 2000.
- [3] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, pages 202–215, Banff, Canada, October 2001.
- [4] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, Canada, August 1987.
- [5] John Douceur. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, Massachusetts, March 2002.
- [6] Freenet. <http://freenet.sourceforge.net/>, 2002.
- [7] Gnutella. <http://gnutella.wego.com/>, 2002.
- [8] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 654–663, El Paso, Texas, May 1997.
- [9] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 569–578, El Paso, Texas, May 1997.
- [10] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In Carl Pomerance, editor, *Advances in Cryptology - Crypto'87*, number 293 in Lecture Notes in Computer Science, pages 369–378. Springer-Verlag, 1987.
- [11] David L. Mills. Network Time Protocol Specification, Implementation and Analysis. Network Working Report RFC 1305, March 1992.
- [12] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, pages 161–172, San Diego, California, August 2001.
- [13] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP-01)*, pages 15–28, Banff, Canada, October 2001.
- [14] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.
- [15] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, pages 188–201, Banff, Canada, October 2001.
- [16] Secure Hash Standard. US Dept. of Commerce/NIST, 1995.
- [17] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, Massachusetts, March 2002.
- [18] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, pages 149–160, San Diego, California, August 2001.
- [19] Ben Y. Zhao, John Kubiawicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, University of California at Berkeley, April 2001.
- [20] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A secure distributed on-line certification authority. Technical report, 2000-1828, Department of Computer Science, Cornell University, December 2000.

Self-Organization in Peer-to-Peer Systems

Jonathan Ledlie, Jacob M. Taylor, Laura Serban, Margo Seltzer
Harvard University

33 Oxford St., Cambridge MA., USA

{jonathan,margo}@eecs.harvard.edu, {jmtaylor,serban}@fas.harvard.edu

Abstract

This paper addresses the problem of forming groups in peer-to-peer (P2P) systems and examines what dependability means in decentralized distributed systems. Much of the literature in this field assumes that the participants form a local picture of global state, yet little research has been done discussing how this state remains stable as nodes enter and leave the system. We assume that nodes remain in the system long enough to benefit from retaining state, but not sufficiently long that the dynamic nature of the problem can be ignored. We look at the components that describe a system's dependability and argue that next-generation decentralized systems must explicitly delineate the information dispersal mechanisms (e.g., probe, event-driven, broadcast), the capabilities assumed about constituent nodes (bandwidth, up-time, re-entry distributions), and distribution of information demands (needles in a haystack vs. hay in a haystack [13]). We evaluate two systems based on these criteria: Chord [22] and a heterogeneous-node hierarchical grouping scheme [11]. The former gives a $> 1\%$ failed request rate under normal P2P conditions and a prototype of the latter a similar rate under more strenuous conditions with an order of magnitude more organizational messages. This analysis suggests several methods to greatly improve the prototype.

1. Introduction

Large-scale decentralized distributed systems — peer-to-peer, ubiquitous, or sensor network systems with multiple millions of nodes — are just beyond their infancy and have not yet had dependability quantified in a consistent manner. Many P2P projects are in their research phases and only a handful are in common use. Most of these designs work well when the rate at which nodes enter and exit the system is small, but none that we have seen explicitly discuss the design tradeoffs between the type of information exchange for which the system is designed, the physical characteristics of

the constituent nodes (e.g., mean time to failure), and how reliable information exchange needs to be in order to fulfill the system's goals. Before more large-scale decentralized distributed systems are designed and built, the community needs to reach agreement on the meaning of acceptable and unacceptable functionality under a variety of dynamic conditions representative of P2P systems.

The intuitive notion of “dependability” for these systems is one of *reachability of information*. Accordingly, dependability should be measured by the percentage of times that a request results in the proper information moving from its source(s) to its destination(s). The requirements for dependability vary greatly within the parameter space of P2P systems. Consider a point-to-point system designed to answer existence queries. An instance where every node has a completely up-to-date and accurate picture of the rest of the system and where the bandwidth consumed by queries and state transfer does not exceed the capacity of any links would be perfectly dependable. However, such a design might not work if the type of information exchanged was event-driven: if, for example, one node needed to notify another node or collection of nodes when there was an abrupt temperature change or if a bridge were about to collapse.

In this paper, we define dependability in P2P systems and discuss the way in which Chord and our own hierarchically grouped system self-organize to overcome the unreliability of nodes that comprise the system.

2. Reliability in Decentralized Systems

A system's dependability is defined in terms of three characteristics: the type and method of information exchange (e.g., probes, point-to-point streams, broadcast streams, etc.), the individual nodes' capabilities, and the distribution of data and queries among the nodes. One thread links all three components: local information must provide a quantifiable and probabilistically accurate depiction of the global state. The required level of this accuracy depends on system usage; increased tolerance for out of date local information leads to diminished state, messages, bandwidth, and

uptime requirements. For example, in Chord, the likelihood that requests will be fulfillable depends on the join/failure rate and on the rate at which a node ring stabilization procedure is run, which in turn depends on the node's capacity for topology messages.

The first component to the overall dependability of a decentralized system is the type of information exchanged across it. We divide information exchange into the following five categories:

probe Probe queries test for the existence of an object.

These queries often use a filter structure (*e.g.*, DHTs or Bloom filters) or resource intensive naive broadcast queries (*e.g.*, Gnutella [8], Freenet [5]); the latter gives the significant advantage of high tolerance against node failure and allows for receiver-interpreted queries.

event-driven point-to-point A node registers an interest and is contacted when something matching this interest enters the system. Examples include abrupt temperature change, sensor aggregators [9], change in file contents, file creation, new authorship, and distributed triggers[3].

event-driven broadcast This is a broadcast from one node to all other nodes, used to distribute information globally. This could be used, for example, to implement a software update.

continuous stream point-to-point This exchange provides a path for streaming data for an indeterminate duration to another node or other nodes. Internet routing is one such example. The requirement of continuity may mean pro-active measures against unknown failures will be necessary (*e.g.* using multiple paths), compared with just post-failure cleanup and recovery.

continuous stream broadcast One node continuously updates the entire system, similar to continuous stream point-to-point. The ubiquitous nature of this type of exchange may make it much easier to implement in P2P systems without pro-active routine measures.

Most P2P designs focus on probe queries while sensor network systems fall into one of the remaining four classes. That said, one could imagine other systems where P2P systems support event-based queries and sensor networks use probes. Regardless, the categories of requests must be considered when defining the system's local state.

The nodes' capabilities are the second characteristic that defines a system's dependability. Liben-Nowell *et al.* introduced the idea of the half-life of a system as the amount of time it takes for half of the nodes to exit [12]. One can generalize this concept to include the probability of the node returning to the system and the length of time between a node's exiting and returning. Most of the current crop of

popular P2P research systems [6, 7] were designed with a large-scale static system in mind, and do not perform well under high volatility. Unfortunately, this high volatility is the norm: a study of Mojo Nation found that 80% of the nodes exist in the system for less than one hour [25]. If they are to be deployed on dynamic P2P networks or on battery-conscious sensor networks, new designs need to incorporate the distribution of node join and return from the beginning.

The third component of dependability is the distribution of information and requests across the nodes. Adar and Huberman [1] and Ripeanu [18] show that a small percentage of files make up the bulk of the queries and files in Gnutella, and that the distribution of searches are heavy-tailed. Lv *et al.* [13] use the correlation between file and query distributions to explain why Gnutella has not collapsed under the weight of its naive, flooding-based probe scheme, as Ritter predicted [19]. The simulation studies of this paper assume a uniform distribution of files and file requests, which is probably not appropriate for the Gnutella world. However, it provides an upper-bound on the query failure rate for a correlated system. Also, Gnutella-like file correlations are clearly inappropriate for a library citation, DNS, or similar environment where users search for both hay and needles in a haystack of information [13]. We see that it is essential to consider the distribution of both data and query elements in order to evaluate the system's dependability.

Terminology from the fault tolerant community can be misleading when applied to distributed decentralized systems. Mean-time-to-failure here refers to the mean-time-to-node-departure. Mean-time-to-data-loss has less meaning when nodes are always entering and exiting the system; queries always have a significant chance of failing under realistic conditions. For P2P filesharing systems we can define mean-time-to-query-failure (MTQF). More generally, the dependability can be quantified by the mean-time-to-request-failure (MTRF), which allows for all five categories of information exchange to be considered.

For Chord and the hierarchical grouping system we describe below, we assume exact probe searches on a uniform distribution of files present among live nodes in the system: only files that currently exist are searched for. We assume that nodes are not overburdened with other activities, *e.g.*, actually moving data. In the Chord experiments, we assume a uniformity of node capabilities — somewhat realistic in a sensor network with uniform components, but unrealistic in a P2P setting. In the hierarchical grouping simulator, nodes can have both uniform and heterogeneous characteristics, and we present results for both of these situations. These assessments are clearly only the beginning to a long series of possible evaluations. It seems unlikely that one system will work well under the whole range of P2P parameters; designers must explicitly state their target node and data audience and then evaluate dependability accordingly.

3. Topology of Hierarchical Groups

We have designed and implemented a simulator for a hierarchical grouping scheme which is designed for P2P and sensor network systems. In earlier work, we evaluated a prototype of its search mechanism [11], but its grouping mechanism has not been previously described. For completeness, we present both here. We refer the reader to other papers [22, 23] for an introduction to Chord.

3.1. Search Overview

We begin by describing how the search system works and then move into details of how the system self-configures. A system consists of many hierarchical groups, each shaped as a tree. Every group has a root node. The root is responsible for:

1. Calculating a summary of all objects in the group.
2. Maintaining summaries for each of its immediate children (which in turn maintain summaries for their children).
3. Directing searches of the group.

Summaries are represented as Bloom filters [4], whose size is computed by the root. Bloom filters are bit strings whose size is proportional to the number of objects they summarize [16]. All bits are initialized to zero; the addition of each object to the filter sets “on” the bits signaled by several hash functions for which the object being added is the input. Bits that are already set remain on. To probe a filter for a match, the same hash functions are performed and the bit array is checked: if all of the bits are set, the filter matches. Bloom filters only give false positives, not false negatives. Nodes underneath the root are arranged in a k -tree structure with $\log_k n$ nodes at level n . Nodes communicate with their children, their parent, with the root of their group, and with a dynamically changing collection of extra-group nodes.

Given this configuration, a search originating at node N , proceeds as follows:

1. Consult the group summary filter of node N . If the filter indicates that the object could exist in the tree, iterate over all possible children that might contain the object (using the child filters stored at N). If the object is found in any child, the search concludes successfully.
2. If the object is not found in the current tree, node N passes the query to $\text{root}(N)$. $\text{root}(N)$ conducts a search on its descendants (pruning the part of the tree already searched by N).
3. If $\text{root}(N)$ fails to locate the object in its current tree, it sends the request to any groups whose filter indicates that the object could reside there.

4. Each group queried takes one of three actions:

- (a) If the current group filter indicates that the object cannot be in the group, the group responds with the new group filter.
- (b) If the object isn't in the group, respond with a NACK and some suggested groups that might be queried (that is, consult any other group summaries present and for any potential hits, tell the initiating group of the potential hit).
- (c) Return the location of the node that has the object in the group.

3.2. Hierarchy Structure

We define the ideal topology as a collection of groups of nodes, where the nodes in a group are related based on low intra-group latency and varied mean-time-to-failure (MTTF), and heterogeneous bandwidth. Our goal is to come as close to this ideal topology as possible using only local information.

Nodes benefit from being in a group because they share information about other groups, so that when node a in G_1 receives information about G_2 , that information is accessible to all other nodes in G_1 , because G_1 's root caches G_2 's filter. These benefits increase as groups grow in size. Nodes also benefit from the existence of other groups, because transmitted group summaries serve as an efficient mechanism to prune the search space.

Larger groups provide more shared information, but this benefit is offset by the cost of keeping the group reasonably balanced, maintaining group summaries, and the load on the root. The root's workload grows with the size of the group as it will broker all group searches, maintain group and child summaries, control entry to the group and determine the time for partitioning of the group.

It is this last responsibility, determining partition time, that makes the system feasible. When the root becomes overloaded, it sheds load by partitioning the group. This partitioning, in conjunction with responding to requests to join the group, is what provides the dynamism and self-configurability of the system.

Several other projects have proposed “supernodes” as the solution to the heterogeneity empirically extant in P2P systems. Saroiu *et al.* have shown that there are multiple, distinct categories of nodes, ranging from always-on high-bandwidth nodes to 56k modems only connected for an hour or less [20]. Hierarchies form a good extension to the “supernodes” currently proposed in several research projects (*e.g.*, Gnutella++ [8], Brocade [26]). In these projects, there are two levels of nodes: “supernodes” that do most of the routing, and regular nodes. A more general heterogeneous

system should use a heterogeneous topology, with “better” nodes living closer to the root of each group.

3.3. Node Entry

1. When a node x enters the system, it immediately becomes its own group G_x . G_x forms a list of credentials, including its bandwidth capabilities and its number of public files.
2. Node x contacts a well-known location, called a “node cacher”, to find other nodes S in the system (similar to Gnutella and Mojo Nation). This bootstrapping component only keeps a list of other nodes that have recently contacted it, also trying to find other nodes in the system. Because this list is the only state it contains, it can be easily replicated, and can pop in and out of existence.
3. Using the nodes S that the new node learns about from the “node cacher,” G_x forwards its credentials to the groups containing $s \in S$, by sending messages to each s , which then forward this information to their roots.
4. Each root that considers G_x valid for entry respond to G_x with its credentials. G_x chooses which group to join by picking the one with the best credentials. If this group agrees, G_x then merges with this group. If it refuses, G_x tries another group.

3.4. Node Exit

We have experimented with two designs for keeping the descendants of a node part of a group when a node exits. In one mechanism, nodes try to maintain knowledge of their siblings and grandparents. This information is sufficient to elect a new leader to take the place of the missing parent and then contact the grandparent to inform it of the new topology, including the summary filter change. The other, lazy mechanism just drops children from a group when their parent dies. This expends fewer topology messages and is the one we use in the simulation.

3.5. Summary Propagation

The root determines the group filter size n by using the number of objects in the tree to estimate how many bits are required to produce a Bloom filter with approximately half of the bits set [16]. The lowest leaf nodes generate filters of size n bits by hashing on their stored objects, turning on these bits in their filters, and sending their filters to their parents. These internal nodes also perform their hashing modulo n bits and logically *OR* their filters together with their children’s filters, and pass these filters up the tree. Finally,

the root node will have the summary of all of the objects in the group, with ideally about half of the bits in its filter set (more than half-full filters tend to give many false positives). The hierarchy of filters also makes it likely that more bits are set higher up in the tree and, conversely, that filters are more sparse — and therefore more accurate and more compressible where bandwidth is less — as searches wind down the tree. This information hierarchy is used both outside the group to determine whether to contact the group at all and within to better direct queries between nodes.

As nodes join and expire from the network, we have them self-form into a communication- and information-based hierarchy based on local information. With global information, nodes would form themselves into ideal groups. With local information, however, they will form themselves into a close approximation of this ideal, forming a continuously maintained “almost-ideal” state.

4. Grouping Analysis

We base our grouping model on natural systems that exhibit self-configuration, driven by particle interactions that lower energy costs when an organized state is realized. Evolution models [2], non-equilibrium phase transitions [24], and crystal facet structure formation [15] among others, all show this behavior, and these ideas have been widely applied to a number of economics and engineering problems.

We derive a node’s cost based on its bandwidth consumption, though a refinement to include latency would be a natural extension of this method. To make grouping decisions, nodes compare their current cost with the cost of being in the other groups of which they are aware. If, by forming a group, two nodes can lower the number of queries they receive and the efficiency of the queries they generate, then a group configuration is more desirable. However, there is an activation cost to form a new group, comprised of the one time cost of distributing filters and reorganizing the tree. If groups only form when the cost of the old state exceeds the sum of the activation cost and the new state’s cost, we can encourage stability. Having groups flit in and out of existence is expensive and is mitigated by this activation cost.

As noted above, the overall cost that each node seeks to minimize is the weighted sum of the bandwidth costs. Bandwidth usage consists primarily of queries and filter updates. We assume nodes have poor knowledge of the system outside their own group, making query estimates difficult. The only reliable computation nodes can perform with regards to the costs outlined above are those local to the group, that is, specific to the filters. We set the individual group filter cost to the fraction of bandwidth consumed by filter messages to total bandwidth:

$$C_f(g) = \frac{f_{msg}(g)}{\tau(g)} \frac{f_{size}(g)}{BW(g)}$$

where $f_{msg}(g)$ is the number of filter messages sent out over the time $\tau(g)$ with average size $f_{size}(g)$, and $BW(g)$ is the total bandwidth of the group. A more sophisticated method of estimating the filter message rate involving a weighted average favoring near-time events would be a natural extension of this model.

The estimate of the combined cost is then

$$C_f(g_1 \cup g_2) = \left(\frac{f_{msg}(g_1)}{\tau(g_1)} + \frac{f_{msg}(g_2)}{\tau(g_2)} \right) \frac{f_{size}(g_1 \cup g_2)}{BW(g_1 \cup g_2)}.$$

From this we can guess that the activation cost should go as

$$H = \frac{\text{size}(g_1 \cup g_2) f_{size}(g_1 \cup g_2)}{t_{\text{filterdist.}}(g_1 \cup g_2) BW(g_1 \cup g_2)}.$$

where $t_{\text{filterdist.}}$ is the average time to redistribute a filter. Assuming these factors are uniform for groups of a given size, the activation cost sets the cost scale of the system.

We can also deduce simple information about the effectiveness of searches from the local group information, such as using connectivity information to determine the quality of searches. We define the search quality factor to be

$$Q_s = a(\text{size}(g_1 \cup g_2)) + b(\text{known nodes}(g_1 \cup g_2))$$

with a and b proportional to the inverse of the number of nodes in the system. In the high bandwidth limit we set a to zero and b to the inverse of the total number of nodes in the system. Then Q_s goes to one when a group is connected to the entire system. However, as we may expect many duplicate files, the advantage of having many members in one's own group and not just connected may be significant; this is represented by non-zero a .

The combined cost function is then

$$\text{Cost} = \alpha C_f + \beta / Q_s$$

with the additional constraint of a hard wall in bandwidth usage, that is

$$C_f < \epsilon$$

where $\epsilon \ll 1$ for most P2P systems where bandwidth should be allocated for file-transfer. The parameters for group formation are then α , β , $\gamma = a/b$, and ϵ . In the limit of $\alpha \gg \beta$, bandwidth consumption is minimized, while $\beta \gg \alpha$ (but reasonable ϵ) allows for quality of search maximization constrained by reasonable bandwidth consumption. Small γ corresponds to an emphasis on highly connected groups, while large γ should tend to favor large groups.

4.1. Analysis of Summary Filters

We present a brief analysis of the probability of a false positive for a file not in the system, or the *false positive rate*. As described above, each root node in the system maintains

an up-to-date Bloom filter representing the files in its group. In addition, root nodes acquire the aggregate filters of other groups. Let g be the number of groups in the system, n the number of distinct files per group, b the number of bits per file used and k the number of independent hash functions used in the Bloom filter data structure. Assuming that hash functions are perfectly random, the theoretical probability of a false positive for a file not in the system, or the system's false positive rate is:

$$g \left(1 - (1 - 1/nb)^{kn} \right)^k \approx g (1 - \exp(-k/b))^k$$

where $p_s = \exp(-k/b)$ is the probability that a specific bit in any of the aggregate bloom filters is still 0. Note that given g , b and n , the number of hash functions k can be optimized to minimize that false positive rate. Namely, taking $k = b(\ln 2)$ yields an optimal false positive rate of $f_s = g(1/2)^k = g(0.6185)^b$.

The false positive rate derived above corresponds to a theoretical upper bound on the fraction of groups contacted per search. Since the factor $(1 - p_s)^k$ decreases exponentially with b , the number of bits allocated per file, for optimal number of hash functions k , so does the fraction of redundant search messages between groups.

One potential source for concern in our design is whether the root nodes will have the capacity to store and keep up-dated filters of even a small fraction of the rest of the system. A rough calculation shows this is possible. If we assume that nodes on average store $100 \approx 2^7$ files (as they do in Gnutella[1]), and that aggregate filters are built using $8 = 2^3$ bits per file (giving a false positive rate of $\approx 2\%$), with 1000 nodes, storing all of the filters takes $2^{10} \times 2^7 \times 2^3 = 2^{20}$ bits = 128 kilobytes of storage and 1 million $\approx 2^{20}$ nodes consumes 128 megabytes, still not an unreasonable amount of storage. Of course, as has been noted above, roots are not required to cache all or even most other groups' filters, so the actual amount stored is only a fraction of these values.

This hierarchical use of Bloom filters is different from the attenuated Bloom filters used in OceanStore [17, 10]. Its usage of using a combined filter to describe a distinct subgroup of neighboring nodes is similar the logical *OR* presented here. However, OceanStore does not have the concept of hierarchies of filters with increasing numbers of bits set or of representative group filters.

4.2. Compressed Bloom Filters

Large sparse Bloom Filters can be greatly compressed. Theoretically, an m -bit filter can be compressed to $mH(p)$ bits where p is the probability that a bit in the filter is 0 and $H(p) = -p \log_2 p - (1 - p) \log_2 1 - p$ is the entropy function. For sufficiently large filters, arithmetic coding guarantees close to optimal compression, so if p is small enough,

$H(p)$ is much smaller than 1, and significant savings in the transmission size can be achieved[14].

The bulk of the update messages consist of sparse filters. In particular, both filters of the nodes in the lower levels of the tree hierarchy of a group and update filters are sparse. Updates to filters can be sent as deltas: indices of which bits to turn on or off, instead of sending the whole filter. For a relatively balanced tree a node at level $i - 1$ has approximately $2^{\log_2 n - i} \approx \frac{n}{2^i}$ subnodes, therefore assuming that each node has roughly the same number of distinct files, the sparseness of the child filters of a node in the tree increases exponentially with the level of the node in the tree.

We have presented a brief description of our P2P system which uses hierarchical groups to prune searches and take advantage of node heterogeneity to improve system stability. Now we return to looking at reliability in Chord and in our system.

5. Reliability in Chord

We evaluated Chord by determining how it mapped into our three characteristics of decentralized system reliability and by modifying a pre-existing simulator [21] to test for these characteristics.

Chord, in its current form, supports probe (existence) queries: if an object exists in the system, it (hopefully) returns a pointer to the node storing that object. Although the primary Chord papers [23, 6] do not explicitly discuss node characteristics, it is designed to run on a P2P network, inferring that the nodes exhibit the previously outlined characteristics [20, 18, 1, 25], in particular, that average node lifetime is on the order of one to several hours. As previously mentioned, the objects chosen to query are chosen randomly from those currently existing on live nodes.

We modified a Chord simulator to count the number of messages used for search, object relocation, and stabilization. The stabilization process provides a mechanism for nodes to confirm they are the predecessors of their successors and to repair their finger and successor tables. This operation keeps the ring intact and the more frequently nodes join and exit, the more frequently `stabilize` needs to be run to keep failure rates level.

We show the results from two sets of experiments in Figures 1 and 2. Both experiments were run with 1000 nodes performing one search per minute on average (all average events for the Chord simulator follow a Poisson distribution). The number of fingers was 40 and the number of successors was 20. The average message delay for Chord and for hierarchical groups was 50 milliseconds. Both figures average five separate experiments (errorbars were omitted from Figure 2 for visual clarity). Figure 1 shows the gradual improvement in the failed lookup rate as the average lifetime increases from 15 minutes to 3 hours. The `stabilize`

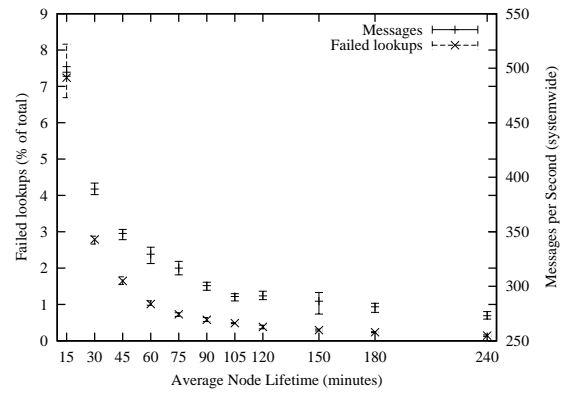


Figure 1. Effect of varying average lifetimes in Chord. Stabilization rate is 2 msg per min per node.

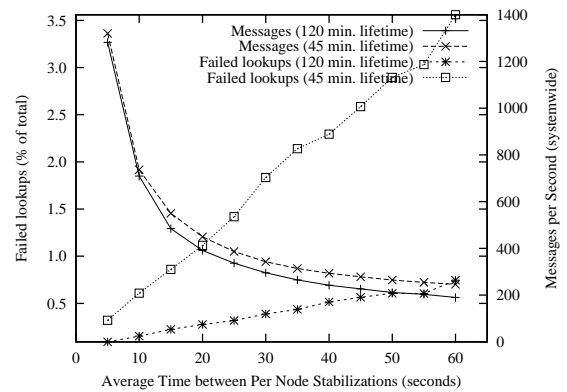


Figure 2. Effect of rate at which stabilizing process is run in Chord.

procedure runs once every 30 seconds per node on average, updating two pointers per run: the successor and a randomly chosen other node from the successor table, with nearby successors chosen with higher probability. This appears to be the default used in most of the experiments in the Technical Report [23], except the one on lookup failure, where the simulator updates all finger entries on every invocation. Each experiment ran for 3 or more hours of virtual time; we found that shorter experiments yielded results with higher variability.

Figure 2 shows Chord as the rate of stabilization changes with average lifetimes of 45 minutes and 120 minutes. Increasing message rates to ~ 1.25 messages per node per second allows a reduction in failed lookups to $< 0.4\%$.

This means from a reliability standpoint that reaching the levels of availability expected by most file system users, say

> 99.99%, would be difficult in Chord unless (a) all nodes exhibit higher than previously observed levels of uptime or (b) all nodes have the bandwidth capacity to run stabilize many times per second. The preferential availability of some data of others is impossible in Chord due to its DHT design; of course, using a higher layer for redundancy would partially ameliorate the availability problem (as CFS does [6]).

6. Reliability in Hierarchical Groups

We have designed and implemented a simulator prototype whose nodes follow the steps outlined in Sections 3 and 4 to form groups and perform searches. Although we ran tests with larger numbers of nodes, we present results with 1000 nodes up on average and with nodes performing one search per minute on average. Because the parameter space for hierarchical groups is large (there are more than 20 separate parameters once we include different possible network configurations), we have been limited in the variety of experiments we could examine prior to this publication.

The primary simplification made in the current simulator is that groups are formed only virtually. That is, nodes do not walk a branch of the tree and reach a final destination; instead, they all exist directly under the root. The death of a node, however, dislocates nodes following a distribution which is based upon failure of nodes in a tree shaped topology with $\log_2 n$ nodes at level n , such that approximately half of the failures are assumed to be leaf failures, then half of those remaining parents with one child, an so on. This has prevented us from evaluating the effectiveness of intragroup filters and seeing the benefits resulting from the compressed bloom filters at the bottom of the tree.

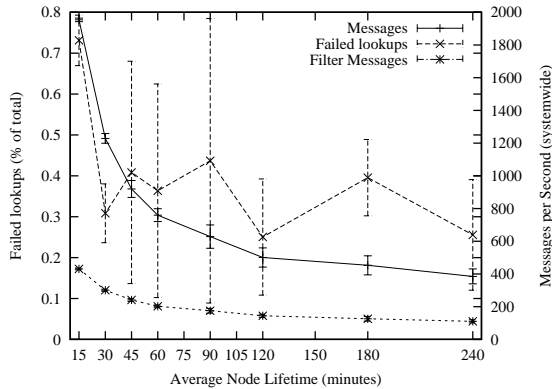


Figure 3. Effect of varying average lifetimes in Hierarchical Groups.

In our simulations we set $\alpha = 0$, $\beta = 1$, and $\gamma = 1/2$, with a maximum filter-message bandwidth usage $\epsilon = 0.1$

and no activation cost. We hope to emphasize quality of searches over bandwidth usage beyond the hard limit of ϵ . In the long-lifetime limit (average node uptimes beyond ten hours) within 10 minutes a 1000 node system stabilized to a better than a 0.1 % search failure system rate. We also ran four simulations per average lifetime using a uniform distribution of nodes with average lifetimes between 15 minutes and 4 hours and 56k modem-like bandwidths. Filter sizes were set to 5 bits per file to minimize bandwidth consumption for filters. One node-cacher was used, disseminating five nodes to every joining node. These results are shown in Figure 3. We see a much lower failure rate and higher message rate than Chord for similar node lifetimes, doing better than 1% even with average lifetimes of 15 minutes. The large fluctuation of results is most likely due to digitization effects from small node number near 0.1% failure rates. Comparing to Figure 2, we see that a similar number of messages are required for both simulators to get similar failure rates at 45 minutes. However, the bulk of these messages are group join queries, followed by filter messages and then all others (search, other intra-group), and that of these the vast majority are generated by node failure leading to child nodes regrouping. In particular, a k -tree leads to deaths creating order $\log_k \text{size}(g)/k$ times the number of nodes disseminated by the node-cacher messages. For the simulations with average lifetimes of 45 minutes, this corresponds to ~ 600 msg/min. We expect that widening the tree or adding grandparent to child links as discussed previously should mitigate this cost. The extra links would require two additional messages per join and k additional messages per death, which is ~ 120 msg/min for an average lifetime of 45 minutes and 1000 nodes. Then, the filter message rates as shown in Figure 3 should dominate.

We also ran simulations with a subset of nodes with high-bandwidth and long-uptimes, as might be expected in an actual network, to look for the effects of these nodes in the trees. They were given 10 times the bandwidth and lifetime of the regular nodes. However, with only 5% of nodes the high-bandwidth type, we see little improvement at the level of 1000 nodes, though larger systems may demonstrate this more effectively. Furthermore, as group rearrangement and root replacement have yet to be implemented, the benefit of these extra nodes may be marginal.

By introducing these refinements and in addition considering sparse filter compression, we can make the limiting factor for group overhead filter messages, and reduce the current bandwidth consumption by filters by a factor of 80% for groups of size 200 and 96% for groups of size 1000. Thus we can scale from a maximum group size of 200 in our simulation for $\epsilon = 0.1$ and MTTF of 45 minutes to well beyond that for even smaller ϵ . It seems that root node over-usage will become the limiting factor for large numbers of nodes. As shown earlier, this will allow for scaling the sys-

tem to order 10^6 nodes.

7. Conclusion

This paper makes three contributions. First, we examine how the implicit goals and assumptions about a particular decentralized system affect measures of its reliability. Second, we introduce a self-organizing hierarchically-based P2P system. Third, we take the assumptions implicit in current P2P filesharing systems and evaluate the reliability of Chord and the hierarchical grouping system. In simulation experiments, both systems perform adequately as long as there exist a 0.5 – 3% tolerance for failure under normal conditions. This failure rate is probably acceptable for a file sharing situation but would need to be tampered by a higher-level application that would provide redundancy in more rigorous file system-like scenarios. Both systems utilize self-configuration — stabilize and local-information-based group formation — to maintain an adequate degree of reliability even under high fluctuation. In particular, our model enables the formation of local points of stability and high bandwidth, and we show how self-configuration can create many local foci to which the rest of the more dynamic system can attach.

We would like to thank M. Mitzenmacher for helpful discussions.

References

- [1] E. Adar and B. Huberman. Free riding on Gnutella. *First Monday*, 5(10), October 2000.
- [2] P. Bak and K. Sneppen. Punctuated equilibrium and criticality in a simple model of evolution. *Physical Review Letters*, 71:4083, 1993.
- [3] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman. The asilomar report on database research. *ACM SIGMOD Record*, December 1998.
- [4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [5] I. Clarke. Freenet: A distributed anonymous information storage and retrieval system. <http://freenetproject.org/cgi-bin/twiki/view/Main/ICSI>, 2001.
- [6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.
- [7] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.
- [8] Gnutella. Gnutella protocol specification v0.4. <http://www.clip2.com/GnutellaProtocol04.pdf>, 2001.
- [9] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.
- [10] J. Kubiawicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM AS-PLOS*. ACM, November 2000.
- [11] J. Ledlie, L. Serban, and D. Toncheva. Scaling filename queries in a large-scale distributed file system. Research Report TR-03-02, Harvard University, January 2002.
- [12] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Observations on the dynamic evolution of peer-to-peer networks. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, March 2002.
- [13] Q. Lv, S. Ratnasamy, and S. Shenker. Can heterogeneity make Gnutella scalable? In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, March 2002.
- [14] M. Mitzenmacher. Compressed Bloom filters. In *Twentieth ACM Symposium on Principles of Distributed Computing (PODC 2001)*, 2001.
- [15] C. Perez, A. Corral, A. Diaz-Guilera, K. Christensen, and A. Arenas. On self-organized criticality and synchronization in lattice models of coupled dynamical systems. *International Journal of Modern Physics B*, 10:1111, 1996.
- [16] M. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237–1239, October 1989.
- [17] S. Rhea and J. Kubiawicz. Probabilistic location and routing. In *INFOCOM 2002*, January 2002.
- [18] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings of International Conference on Peer-to-peer Computing*, August 2001.
- [19] J. Ritter. Why Gnutella can't scale. <http://www.darkridge.com/~jpr5/doc/gnutella.html>, 2001.
- [20] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2002.
- [21] I. Stoica. Chord simulator. <http://www.fs.net/cvs/sfsnet/simulator/?cvsroot=CFS-CVS>, 2001.
- [22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.
- [23] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Research report, MIT, January 2002.
- [24] P. Weichman, A. Harter, and D. Goodstein. Colloquium: Criticality and superfluidity in liquid 4He under nonequilibrium conditions. *Reviews of Modern Physics*, 73:1, 2001.
- [25] B. Wilcox-O'Hearn. Experiences deploying a large-scale emergent network. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, March 2002.
- [26] B. Y. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. D. Kubiawicz. Brocade: Landmark routing on overlay networks. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, March 2002.

HiScamp: self-organizing hierarchical membership protocol

Ayalvadi J. Ganesh, Anne-Marie Kermarrec and Laurent Massoulié

Microsoft Research

7 J J Thomson Avenue

Cambridge CB3 0FB, UK

{ajg, annemk, lmassoul}@microsoft.com

Abstract

Gossip-based or epidemic algorithms rely on a peer-to-peer model for dissemination of multicast messages, and are simple, scalable and reliable. However, traditional gossip-based protocols suffer from two major drawbacks: (i) they rely on each peer having knowledge of the global membership and (ii) they are oblivious to the underlying network and impose a high load on core router links. In this paper we present a self-organizing hierarchical membership protocol which attempts to solve these two issues. Nodes organize themselves into clusters reflecting the topology, and obtain partial views of the membership both within and outside the cluster. The size of the partial views is tuned automatically to achieve high reliability. Gossip messages are targeted mainly within clusters, thereby reducing network load.

1 Introduction

The lack of deployment of IP multicast has led to interest in application-level multicast [11]. Centralized or partially-centralized approaches, proven efficient in local-area networks [12], do not scale to large size groups. Recently some application-layer multicast protocols [3, 16] have been deployed on top of large-scale peer-to-peer routing infrastructures. These approaches have proven to be efficient and scalable but require the existence of such a peer-to-peer infrastructure.

In this context, gossip-based protocols [1, 4, 6] have been shown to be both scalable and reliable. This class of protocols was first introduced to manage consistency in replicated databases [4, 8] but also serves other purposes such as failure detection [18], garbage collection [5] or more recently system management [17].

These protocols rely on epidemic style dissemination, and have several attractive features: (i) they are scalable, (ii) they are highly resilient to failures and (iii) they do not

require the maintenance of global state information. Their scalability relies on a peer-to-peer interaction model and reliability on the use of redundant messages: they use randomization and redundancy to fight random unreliability (of nodes and network links) in a distributed system. In gossip-based protocols, messages are propagated as follows. When a node generates a message, it sends it to a randomly chosen subset of other nodes. When any node receives a message for the first time, it does the same. The number of gossip targets is called the fan-out. Many variants of gossip-based protocols exist, which differ in the number and choice of gossip targets. Consider first the case when gossip targets are chosen uniformly at random from among all group members. It has been shown [13] that, if there are N members in the group, then the fanout required to ensure that the message reaches every group member (with high probability) is of the order of $\log(N)$. Moreover, with such a choice of fanout, the number of hops a message traverses to reach an arbitrary group member is of the order of $\log(N)$ [15]. Therefore, the load on each group member and the latency of multicast delivery increase only logarithmically with the group size. In addition, performance degrades gracefully in the presence of node failures and message losses.

Two main factors limit the applicability of gossip in large-scale, wide-area settings. First, traditional gossip protocols rely on each node employing full knowledge of the group membership in choosing gossip targets. This has motivated work on distributed algorithms [6, 14] to provide each node with a partial view of the membership that is adequate to achieve high reliability. However, even if each node has a partial view, some coordination is required to set and update the fanout value as the size of the system changes. Second, existing gossip protocols are oblivious to the underlying network topology and hence impose a high load on core routers in wide-area systems.

Recently hierarchical epidemic protocols have been proposed to limit the load on core routers in the networks [9, 17]. In this paper, we present HiScamp, a hierarchical self-organizing membership protocol for gossip-based dis-

semination which addresses these issues. HiScamp uses a distance measure to dynamically cluster the nodes in a hierarchy. At each level of the hierarchy, it implements an instance of Scamp [7], a peer-to-peer membership protocol which automatically tunes the size of partial views to the logarithm of the system size. By targeting most gossip messages within clusters, the load on core routers is reduced without compromising reliability. We evaluate HiScamp on a simulator using the Georgia Tech [19] transit-stub model. We compare HiScamp (hierarchical gossip) to Scamp (flat gossip) in terms of link stress and reliability. Preliminary results show a great reduction in network load at the cost of a small decrease in reliability and small increase in latency.

2 Fanout and reliability in gossip-based protocols

We begin by briefly reviewing the relation between fanout and reliability for gossip-based protocols. These results are used to parametrize the membership schemes so that they provide partial views that are large enough to ensure high reliability.

2.1 Flat gossip

Let N denote the number of nodes in the system and suppose each node receiving a message gossips it to K other nodes, chosen uniformly at random among all the nodes. It was shown in [13] that, if the fanout K is $\log(N) + b$, for an arbitrary constant b , then the probability that a gossip message reaches all nodes goes to $\exp(-\exp(-b))$ as N grows large.¹ Note that this refers, not to the probability that a given node receives the message, but to the probability that *every node* receives it. Traditionally, atomic multicast verifies the “all or nothing” property, whereas the notion of atomicity in this paper refers to the “all” property. This result implies that there is a sharp threshold at $\log(N)$; the probability that a gossip message reaches all nodes is close to 1 if each node gossips to slightly more than $\log(N)$ other nodes, and close to 0 if it gossips to slightly fewer than $\log(N)$ other nodes. Table 1 below shows the relationship between b and the quantity $1 - e^{-e^{-b}}$, which is the probability of failing to reach at least one node. By choosing the design parameter b appropriately, we can provide a suitable probabilistic reliability guarantee. The result can be extended to account for node and link failures [13].

¹The same result applies if the fanout is itself a random number, with mean $E[K] = \log(N) + b$, under some mild additional conditions on the distribution of K .

2.2 Hierarchical gossip

In this model, nodes are grouped into clusters, and each node can gossip to other nodes either within its cluster or in other clusters. The question is how to choose the number of gossip targets within the cluster for each node (intra-cluster fanout, denoted k), and the total number of gossip targets outside the cluster aggregated over all nodes in the cluster (inter-cluster fanout, denoted f). In order for gossip to succeed in this model, i.e., for a gossip message to reach all nodes, it is sufficient that the message goes from its originating cluster to every other cluster and that gossip also succeeds within each cluster. Let there be M distinct clusters with n nodes in each cluster and let $N = Mn$ denote the total number of nodes. We want to choose the intra-cluster fanout k and the inter-cluster fanout f in order to guarantee bounds on the probability of success. It was shown in [13] that, if $f = \log(M) - \log(\beta/2)$ and $k = \log(N) - \log(\beta/2)$ for some parameter $\beta > 0$, then the probability that a gossip message reaches all nodes is at least $e^{-\beta}$. Thus, it is sufficient to take the inter-cluster fanout to be logarithmic in the number of clusters, and the intra-cluster fanout to be logarithmic in the total number of nodes (not just the nodes within that cluster). These estimates can be modified to account for node and link failures as in the case of flat gossip, and can also be extended to a multi-level hierarchy of clusters.

3 Hierarchical membership protocol

The objective is to develop a fully decentralized and self-organising membership protocol which ensures that each group member has a partial view of the membership of a size adequate to support gossip reliably (i.e., with reliability comparable to traditional schemes employing full knowledge of group membership). These partial views should be created in a peer-to-peer fashion without the use of servers, and their size should adapt automatically to changes in the size of the group as members join and leave. We described in earlier work [7] how these objectives can be met in order to support flat gossip. We proposed a self-organizing membership protocol called Scamp, which provides each node with a partial view of the membership. The size of this view scales automatically as $c \log(N)$, the multiple c being a design parameter. The partial view is used to propagate gossip messages, and is of the right size to ensure high reliability. The choice of c depends on the degree of resilience to failures required; atomic multicast can be achieved with high probability if the proportion of failed nodes or links is strictly smaller than $(c - 1)/c$. The partial views generated by Scamp do not take locality into account. We wish to augment the basic protocol to support cluster-based gossip, as described above. When nodes are organized into clus-

b	-2	0	2	4	6	8	10
$1 - e^{-e^{-b}}$	0.999	0.632	0.127	0.018	0.002	3E-4	5E-5

Table 1. Dependence on b of probability of non-atomic multicast

ters using topological information, we need to provide each node with two partial views, one of nodes within the same cluster, and another of nodes in other clusters. For clarity of exposition, we describe a protocol with just two levels in the hierarchy. The scheme can be extended in an obvious way to handle hierarchies with more than two levels. We begin by describing the basic Scamp protocol and then show how it can be modified to maintain a hierarchy of views.

3.1 Protocol overview of Scamp

Scamp is a scalable membership protocol which operates in a fully decentralized manner and provides each group member with a partial view of the group membership. The size of the partial views naturally converges to the value required to support a gossip algorithm reliably. The protocol consists of mechanisms for nodes to subscribe (join) and unsubscribe (leave) from the group, and for nodes to detect and recover from isolation. The partial views at nodes evolve in response to changing group membership in a fully decentralised way.

The subscription algorithm proceeds as follows:

1. **Contact** New nodes join the group by sending a subscription request to an arbitrary pre-existing member, called a *contact*. They initialize their partial view with the *nodeId* of the contact.
2. **New subscription** When a node receives a new subscription request, it forwards the new *nodeId* to all members of its own partial view. It also forwards $c - 1$ additional copies of the new *nodeId* (c is a design parameter that determines the proportion of failures tolerated) to randomly chosen nodes in its partial view.
3. **Forwarded subscription** When a node receives a forwarded subscription, it integrates the new subscriber in its partial view with a probability p which depends on the size of its view. If it doesn't keep the new subscriber, it forwards the subscription to a node randomly chosen from its local view. These forwarded subscriptions may be kept by the neighbours or forwarded, but are not destroyed until some node keeps them.
4. **Keeping a subscription** Each node maintains two lists, a *PartialView* of nodes it sends gossip messages to, and an *InView* of nodes that it receives gossip messages from, namely nodes that contain its *nodeId* in

their partial views. If a node i decides to keep the subscription of node j , it places the *id* of node j in its partial view. It also sends a message to node j telling it to keep the *nodeId* of i in its *InView*.

Observe that this subscription protocol only requires local information available at the node treating the subscription request. It is shown in [7] that the system configures itself towards views of size $c \log N$ on average, where N is the total number of nodes in the system. Performance evaluation has shown that multicasting done on top of Scamp exhibits a similar degree of reliability as traditional gossip-based schemes which requires each member to maintain the list of all group members.

The unsubscription mechanism works as follows. Assume the unsubscribing node, say node n_0 , has ordered the *id*'s in its partial view as $i(1), \dots, i(\ell)$, and the *id*'s in its *InView* as $j(1), \dots, j(\ell')$. The unsubscribing node will then inform nodes $j(1), j(2), \dots, j(\ell' - c - 1)$ to replace its *id* with $i(1), i(2), \dots, i(\ell' - c - 1)$ respectively (wrapping around if $\ell' - c - 1 > \ell$). It will simply inform nodes $j(\ell' - c), \dots, j(\ell')$ to remove it from their list but without replacing it by any node *id*. This protocol remains local and only the unsubscribing node and its direct neighbours in the graph are involved in the unsubscription process. It is shown in [7] that this unsubscription mechanism preserves the scaling relation between view and system sizes.

3.2 HiScamp protocol description

We now describe how to organise nodes into clusters to reflect the network topology, and how to modify the membership protocol above to maintain hierarchically structured partial views of the membership. These partial views can be used to support the hierarchical version of gossip described above.

Let $D(a, b)$ denote some agreed measure of distance between nodes a and b , which each of them can assess. For instance, it could be the round-trip time between them on the Internet, measured using ping, or the number of hops on the path between them. Alternatively, it could be some measure of the cost of this path, such as the available bandwidth, which could be measured using *pathchar* [10] or *packet-pair* [2]. HiScamp dynamically builds a hierarchy of clusters of nodes based on $D(\cdot, \cdot)$, using Scamp at each level in the hierarchy. We illustrate the protocol using a 2-level hierarchy for simplicity.

Each node maintains two partial views, organized hierarchically. The first, called *hview*, has 2 levels (or as many levels as there are in the hierarchy): level 1 specifies the nodes to gossip to within the cluster, and level 2 specifies gossip targets in other clusters. The second partial view, called *iview*, has 1 level (or one fewer than the number of levels in the hierarchy) and specifies the inter-cluster view of the cluster to which this node belongs. In other words, the *iview* of a node is the union of the level 2 *hviews* of all the nodes in its cluster. Thus, the *hview* is specific to a node, while the *iview* is common to all nodes in the same cluster. The *iview* is not used directly for gossiping but only for handling subscriptions. We now describe how the nodes are grouped into clusters, and how these views are created and maintained.

The subscription algorithm works as follows.

1. A node j joins the system by sending a subscription request to the node s which is closest to it² under the distance $D(j, \cdot)$. To the extent that the actual topology of the group permits efficient clustering, it may be adequate to choose any nearby node rather than the closest, as this would still ensure that the new node joins the right cluster.
2. If the distance $D(j, s)$ is smaller than a preset threshold t , then node s includes j in the cluster to which it belongs. In this case, the subscription is processed using Scamp within this cluster. Thus, as in Scamp, the subscription of j is forwarded to all nodes that s would gossip to at the lowest level, namely the nodes in level 1 of the *hview* of s . In addition, $|iview(s)| + c - 1$ copies of the subscription are also forwarded to randomly chosen nodes in this view, where $|iview(s)|$ denotes the cardinality of the *iview* of s . Each node receiving a forwarded subscription either keeps it, or forwards it to a node in its level 1 *hview*. The decision whether to keep or forward a subscription is made just as in Scamp, but based on the size of the level 1 *hview* of the node. Finally, the views of j are initialized as follows. Its level 1 *hview* consists just of s , its level 2 *hview* is empty, and its *iview* is initialized to be the same as the *iview* of s . The last is achieved by having s send a message to j with its *iview*.
3. If $D(j, s)$ exceeds the threshold t , then node j initiates a new cluster, and its subscription is handled at level 2 in the hierarchy. To do this, node s uses its *iview* to initiate Scamp. Thus, it forwards the subscription to nodes in other clusters as specified by its *iview*, as well as forwarding $c - 1$ additional copies of the subscription to random elements of its *iview*. The *hview* plays no

role. Forwarded inter-cluster subscriptions are treated just like in Scamp, and are either kept or forwarded to other clusters using only the *iview*'s at each step. The decision of whether to keep or forward a subscription is based on the size of the *iview* only. When a node keeps such a subscription, it includes the node-id of the new subscriber in its level 2 *hview* as well as its *iview*. This change is gossiped to all members of its own cluster, so that they can update their *iview*'s accordingly.³ Finally, the level 1 *hview* of j is initialized to be empty, while its level 2 *hview* and its *iview* are both initialized to $\{s\}$.

4. In addition to the partial views, each node maintains a hierarchically organised *Inview* consisting of all nodes that target it for gossip. The level 1 *Inview* of node j consists of all nodes i in the same cluster that contain node j in their level 1 *hview*, and its level 2 *Inview* consists of all nodes i in other clusters that contain node j in their level 2 *hview*.

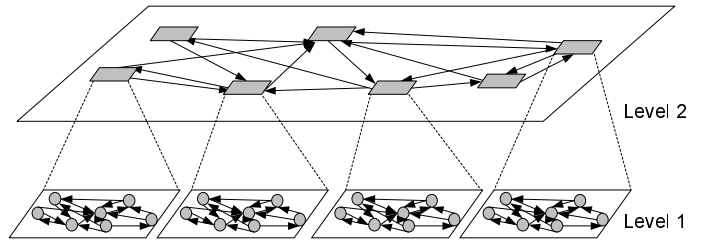


Figure 1. HiScamp overview: each level-2 rectangle represents a cluster in which an instance of the Scamp protocol is implemented; in addition, an instance of the Scamp protocol is implemented at level 2 to connect clusters.

We show in the next section that the sizes of *iview*'s converge automatically to $c \log M$, where M is the number of clusters. Likewise, the sizes of level 1 *hview*'s converge to $c \log N$, where $N = Mn$ is the total number of nodes, and n is the average number of nodes per cluster.

Figure 1 illustrates a two-levels HiScamp protocol. At the lower level, level 1, each cluster implements an instance of the Scamp protocol to connect nodes within a cluster. Each of this cluster is seen as an abstract individual node at the upper level. An instance of the Scamp protocol is used to connect clusters between them at level 2.

In the protocol as described, inter-cluster links only connect the nodes that have initiated each cluster. Such nodes thus represent a single point of failure per cluster, both for

²We assume that a mechanism for providing the Id of the closest node is available; in fact, this is currently an open problem.

³This is done so that all nodes in the same cluster always have the same *iview*. Since *iview* updates are rare, the amortized cost of synchronizing *iview*'s is not large.

messages to reach the cluster, or to be sent outside the cluster. In order to avoid this, we use an algorithm in the background which balances the level 2 *hview*'s so as to ensure that inbound or outbound messages are not handled by a unique node. Therefore, the inter-cluster links are handled by different nodes within a cluster.

This is achieved as follows. Any node whose level 2 *hview* contains more than one element periodically attempts to hand over one of its external links to another member of its cluster. To do this, it removes a node from its level 2 *hview* and forwards the *nodeId* to a random element of its level 1 *hview* (this action is taken only if the level 1 *hview* is non-empty). The node receiving this message treats it as a forwarded subscription; it keeps it with a probability depending on the size of its level 2 *hview* and forwards it if it is not kept. Since the number of inter-cluster links is usually small ($\log M$ aggregated over all nodes within a cluster), the level 2 *hview* of a node will typically be empty or consist of just one node, except in very small clusters. A similar hand-off mechanism is used periodically by any node whose level 2 *Inview* consists of more than one element.

The unsubscription mechanism is exactly analogous to Scamp. Unsubscribing nodes ensure that *nodeIds* at each level of their partial views are transferred to nodes in the corresponding level of their *Inview*.

It is possible that nodes leave a group without implementing the unsubscription mechanism described. A lease mechanism to time out subscriptions and force re-subscription was proposed in [7] to deal with this problem. The same mechanism can be implemented in HiScamp.

4 Performance evaluation

This section presents preliminary results of simulation experiments to evaluate the performance and properties of HiScamp as compare to Scamp. We evaluate and compare the hierarchical and flat protocols according to the following metrics: (i) the impact on the network in terms of link stress; (ii) resilience to node failures; and (iv) the latency of message delivery.

4.1 Experimental setting

We evaluated HiScamp through simulations using network topologies generated randomly according to the Georgia Tech [19] transit-stub model. The topology used in this paper is composed of a 600 node core, with a LAN attached to each core node. Each LAN has a star topology and is composed of 100 nodes on average. Thus, there are 60000 LAN nodes; we selected 50000 of these at random to compose our group. Link delays are modeled simply by assigning a propagation delay (1ms to each LAN link and 40.5ms to each core link). We evaluate the impact of HiScamp by

measuring the stress on each link of the simulated network, *i.e.* the number of messages travelling along that link. This metric provides a good approximation of the load imposed on the network.

4.2 Impact on the network

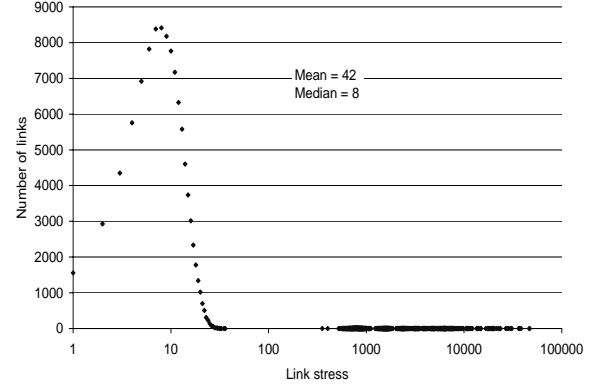


Figure 2. Distribution of link stress in a 50000 node group using Scamp

The main impact expected from HiScamp is to decrease the load on core router links.

In order to evaluate this, we compare the load on each physical link, called the link stress, in Scamp and HiScamp. In our experiments, we broadcast one message, and the link stress is defined as the number of copies of this message that traverse the link. We evaluate the link stress in a non-faulty environment.

Figures 2, 3 and 4 display the distribution of link stress on core router links for the dissemination of one multicast message in an overlay network constructed using HiScamp with 1, 2 and 3 levels respectively. The 1-level configuration is equivalent to a flat Scamp which doesn't take network topology into account. As expected, the average link stress is approximately $\log(N)$ and the maximum link stress is very high since nodes choose gossip targets uniformly, regardless of their physical location. In the 2-level configuration, we set the distance threshold t to 3, which means that nodes within the same LAN (delay 2) belong to the same cluster. In the 3-level configuration, we set distance thresholds at $t_1 = 3$ and $t_2 = 150$. These thresholds were fixed by observing the distribution of delays in typical 10000 node configurations. Detection and dynamic setting of thresholds is an interesting issue that we defer at this point of our study.

The maximum link stress decreases from 46738⁴ in the

⁴This means that almost every node gossips remotely, using a core router link.

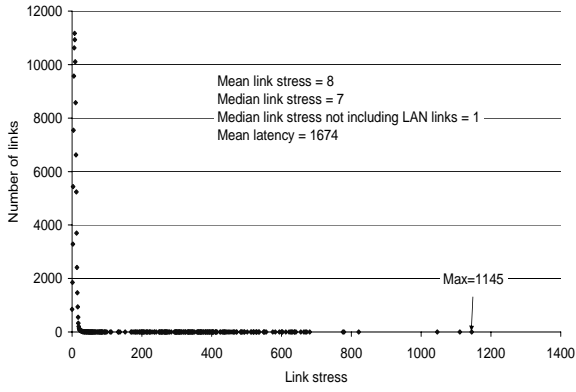


Figure 3. Distribution of link stress in a 50000 node group using a two-levels HiScamp

flat setting to 1145 with 2 levels and to 521 with 3 levels. In both configurations, the medians (with and without including the LAN links) indicate that the network traffic mostly occurs within LANs. In the 2-level configuration, 952 clusters were built dynamically with 18 nodes per cluster on average. In the 3-level configuration, 488 clusters were built at the lowest level and 107 at the top level. In other words, there were about 4 level-2 clusters on average in each level-3 cluster. The results show that clustering significantly reduces the stress on core router links. The main impact of increasing the number of levels in the hierarchy is to significantly reduce the maximum load on a few core router links. In both HiScamp configurations, if we consider 99% of the links, the maximum link stress is 18 (instead of 1145 and 521 if we consider all links).

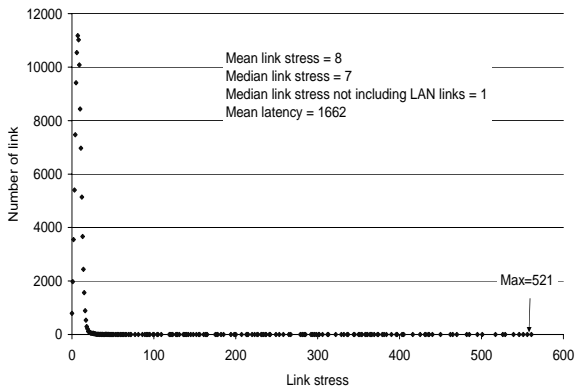


Figure 4. Distribution of link stress in a 50000 node group using a three-levels HiScamp

4.3 Reliability

As mentioned earlier, the performance of gossip-based protocols degrades gracefully in the presence of failures. Another important metric to evaluate HiScamp against is the reliability it provides in comparison to a flat implementation of Scamp. We evaluate the impact of failures, ranging from 10 to 30% of group members, on the proportion of nodes reached by a broadcast message. We consider here a fail-stop model where faulty nodes do not gossip messages they receive. The table below shows reliability figures, expressed as the fraction of non-faulty nodes reached by a multicast message, against the number of node failures. Results show that Hi-Scamp has a lower resilience to failure than a flat implementation. We are currently working on this issue.

% of faulty nodes	0	10	20	30
% of nodes reached in Scamp	100	99.76	99.4	98.77
% of nodes reached in HiScamp (2-levels)	100	97.39	93.64	88.99

Table 2. Comparison of reliability in HiScamp and Scamp in a 50000 node system

4.4 Latency

We also measured the average latency of delivery of one multicast message from the source to every group member: group members experience an average delay of 914 with Scamp, of 1674 with HiScamp with 2 levels and 1662 with HiScamp with 3 levels. So, increasing the number of levels in the hierarchy has an impact on latency but achieves a substantial reduction in the maximum link stress.

5 Conclusion

In this paper, we presented HiScamp, a peer-to-peer hierarchical membership protocol for gossip-based group communication. HiScamp is fully decentralized and leverages the self-organizing properties of Scamp, a scalable membership protocol. Experiments using the Georgia Tech topology model show that, by exploiting locality, HiScamp achieves a much smaller link stress than Scamp, but at the expense of slightly increased latency and degraded reliability. This effect is amplified as the number of hierarchical levels in HiScamp is increased.

The network stress/reliability trade-off needs further investigation, as does the automated selection of thresholds and number of hierarchical levels for cluster creation.

Acknowledgments

We would like to thank Miguel Castro and Antony Rowstron for their substantial role in the development of the simulator.

References

- [1] K.P. Birman, M. Hayden, O.Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM TOCS*, 17(2):41–88, May 1999.
- [2] J.-C. Bolot. End-to-end packet delay and loss behavior in the internet. In *Proceedings of SIGCOMM*, 1993.
- [3] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure, To appear. *IEEE Journal on Selected Areas in communications (JSAC)*.
- [4] A.J. Demers, D.H. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *PODC*, pages 1–12, Vancouver, Canada, August 1987.
- [5] K. Guo et al. Gsgc: an efficient gossip-based garbage collection scheme for scalable reliable multicast. Technical Report TR-97-1656, Cornell University, Department of Computer Science, 1997.
- [6] P.T. Eugster, R. Guerraoui, S.B. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *IEEE Intl. Conf. Dependable Systems and Networks (DSN)*, 2001.
- [7] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Networked Group Communication Workshop (NGC)*, volume 2233 of *Lecture Notes in Computer Science (LNCS)*, pages 44–56, London, UK, Nov 2001.
- [8] R. Golding and K. Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, UC Santa Cruz, Dept. of Computer Science, 1992.
- [9] I. Gupta, A.-M. Kermarrec, and A.J. Ganesh. Adaptive and efficient epidemic-style protocols for reliable and scalable multicast. In *Submitted to publication*, <http://research.microsoft.com/camdis/gossip.htm>, 2002.
- [10] V. Jacobson. Pathchar – a tool to infer characteristics of internet paths. April 1997.
- [11] J. Jannotti, D.K. Gifford, K.L. Johnson, M.F. Kaashoek, and J.W.O'Toole. Overcast: Reliable multicasting with an overlay network. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, Paradise Point Resort, San Diego, California, USA, October 23-25 2000.
- [12] F Kaashoek, a. Tanenbaum, A.S. Hummel, and H.E. Bal. An efficient reliable broadcast protocol. *Operating System Review*, 23, 1989.
- [13] A.-M Kermarrec, L. Massoulié, and A.J. Ganesh. Probabilistic reliable dissemination in large-scale systems. available at <http://research.microsoft.com/camdis/gossip.htm>.
- [14] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide-area network. Technical Report CS1999-0622, University of California, San Diego, Computer Science and Engineering, June 1999.
- [15] B. Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47:213–223, 1987.
- [16] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *Proceedings of the Third International Workshop on Networked Group Communication*, November 2001.
- [17] R. van Renesse and K.P. Birman. Scalable management and data mining using astrolabe. In *IEEE International workshop on Peer-to-peer systems (IPTPS)*, 2002.
- [18] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *International conference on Distributed Platforms and Open Distributed Processing ((IFIP)*, 1998.
- [19] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an inter-network. In *INFOCOM96*, 1996.

One Ring to Rule them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks

Miguel Castro
Microsoft Research,
7 J J Thomson Close,
Cambridge,
CB3 0FB, UK.
mcastro@microsoft.com

Peter Druschel
Rice University,
6100 Main Street,
MS-132, Houston,
TX 77005, USA.
druschel@cs.rice.edu

Anne-Marie Kermarrec
Microsoft Research,
7 J J Thomson Close,
Cambridge,
CB3 0FB, UK.
annemk@microsoft.com

Antony Rowstron
Microsoft Research,
7 J J Thomson Close,
Cambridge,
CB3 0FB, UK.
antr@microsoft.com

One Ring to rule them all. One Ring to find them. One Ring to bring them all. And in the darkness bind them. *J.R.R. Tolkien*

Abstract

Self-organizing, structured peer-to-peer (p2p) overlay networks like CAN, Chord, Pastry and Tapestry offer a novel platform for a variety of scalable and decentralized distributed applications. These systems provide efficient and fault-tolerant routing, object location, and load balancing within a self-organizing overlay network.

One major problem with these systems is how to bootstrap them. How do you decide which overlay to join? How do you find a contact node in the overlay to join? How do you obtain the code that you should run? Current systems require that each node that participates in a given overlay supports the same set of applications, and that these applications are pre-installed on each node.

In this position paper, we sketch the design of an infrastructure that uses a universal overlay to provide a scalable infrastructure to bootstrap multiple service overlays providing different functionality. It provides mechanisms to advertise services and to discover services, contact nodes, and service code.

1. Introduction

Recent systems such as CAN [11], Chord [15], Kademlia [8], Pastry [12] and Tapestry [17] provide a self-organizing structured peer-to-peer (p2p) overlay network that can serve as a substrate for large-scale peer-to-peer applications. One of the abstractions that these systems can provide is a scalable, fault-tolerant distributed hash table (DHT), in which any item can be located within a bounded number of routing hops, using a small per-node routing table.

In these systems a live node in the overlay to each key and provide primitives to send a message to a key. Messages are routed to the live node that is currently responsible for the destination key. Keys are chosen from a large space and each node is assigned an identifier (*nodeId*) chosen from the same space. Each node maintains a routing table with *nodeIds* and IP addresses of other nodes. The protocols use these routing tables to assign keys to live nodes. For instance, in Pastry, a key is assigned to the live node with *nodeId* numerically closest to the key.

In the simplest case, DHTs can be used to store key-value pairs much like centralized hash tables. Lookup and insert operations can be performed in a small number of routing hops. The overlay network is completely self-organizing, and each node maintains only a small routing table with size constant or logarithmic in the number of participating nodes. Structured p2p overlays can be used as a platform for a variety of distributed services, including archival stores [7, 3, 13], content distribution [6] and application-level multicast [18, 2, 14].

Service advertisement, discovery and binding are common problems in distributed systems [10, 16, 9]. Service advertisement and discovery mechanisms allow a user to deploy and find services of interest, and binding provides the user with the code necessary to install the service on a node. These problems are compounded in p2p overlays because the service is run by a large number of diverse, distributed peers. Furthermore, binding is harder for a p2p service because a joining peer is required to know a *contact node* already in the overlay.

Current p2p overlays do not provide a good solution to these problems. They require that each node supports the same set of applications, and that these applications are pre-installed on each node. Additionally, they do not provide a scalable solution to find a contact node to join an overlay.

In this position paper, we sketch the design of an in-

frastructure that uses a universal p2p overlay to provide scalable mechanisms to bootstrap multiple service overlays providing different functionality. It provides mechanisms to advertise and discover services, contact nodes, and service code.

In the following description, we will use Pastry as an example structured p2p overlay protocol. It should be noted that the ideas and concepts apply equally to other protocols like Chord, CAN and Tapestry.

2. Pastry overview

In Pastry, keys and nodeIds are 128 bits in length and can be thought of as a sequence of digits in base 16. Pastry routes a message to the node whose nodeId is numerically closest to the key, in a circular nodeId space, which we call a *ring*.

Each node maintains both a leaf set and a routing table. The leaf set contains the immediate L clockwise and counter-clockwise neighboring nodes in the circular nodeId space. A node's routing table is organized into 32 rows and 16 columns. The 16 entries in row n of the routing table refer to nodes whose nodeIds share the first n digits with the present node's nodeId; the $n + 1$ th nodeId digit of a node in column m of row n equals m . The column in row n corresponding to the value of the $n + 1$'s digits of the local node's nodeId remains empty. NodeIds are chosen randomly with uniform probability from the set of 128-bit strings. As a result, only $\log_{16} N$ rows are populated in a node's routing table on average, if there are N nodes participating in the overlay. Figure 1 depicts an example routing table.

In a normal routing step, a Pastry node forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's id. Such a node must exist in the leaf set, unless all of the members in one half of the leaf set have failed concurrently. Given that nodes with adjacent nodeIds are highly unlikely to suffer correlated failures, the probability of this event can be made very small even for modest values of L . The expected number of routing hops is only $\log_{16} N$. Figure 2 shows an example.

Each service is assigned a unique *service id*. When a node determines that it is numerically closest to the key (using the leaf set), it delivers the message to the local service whose service id matches that contained in the message. Moreover, the service is notified on each intermediate node that a message encounters along its route. Services use this to perform dynamic caching, to construct

multicast trees, etc.

Pastry is fully self-organizing. A node join protocol ensures that a new node can initialize its leaf set and routing table, and restore all system invariants by exchanging $O(\log N)$ messages. In the event of a node failure, the invariants can likewise be restored by exchanging $O(\log N)$ messages. Like all other p2p overlays, Pastry requires a *contact node* already in the overlay to bootstrap the join protocol.

Pastry constructs the overlay network in a manner that is aware of the proximity between nodes in the underlying Internet. As a result, one can show that Pastry achieves an average delay penalty, i.e., the total delay experienced by a Pastry message relative to the delay between source and destination in the Internet, of only about two [1].

3. The universal ring

Our infrastructure for service discovery and binding relies on a *universal ring*, which is an overlay that all participating nodes are expected to join. The universal ring only provides services to bootstrap other services. Other services typically form separate overlays, which are created dynamically. The nodes in the service specific overlays are a subset of the nodes in the universal ring. The universal ring enables peers to advertise and discover services of interest, to find the code they need to run to participate in a particular service overlay, and to find a contact node to join the service overlay.

3.1. Joining the universal ring

To join the universal ring, each node needs to obtain a nodeId that is assigned by some element of a set of trusted authorities, e.g., ICANN or a certification authority like Verisign. The certification authority assigns a random nodeId to the node and signs a *nodeId certificate* that binds the nodeId with a public key for a bounded amount of time. The node knows the private key that corresponds to this public key to authenticate itself to other nodes in the overlay. The certification authority should charge nodes for the certificates it issues to make it more difficult for an attacker to control many virtual nodes in the universal ring [4].

After obtaining a nodeId certificate, a joining node needs to obtain the address of a contact node in the universal ring. If a large fraction of the nodes in the Internet are in the universal ring, one can use brute-force, distributed techniques to find a contact node. For example, expanding ring IP multicast or other forms of controlled flooding will work well because they will find a contact node within a few hops of the joining node. Otherwise, servers with well-known domain names can be used, which provide a

0	1	2	3	4	5		7	8	9	a	b	c	d	e	f
x	x	x	x	x	x		x	x	x	x	x	x	x	x	x
6	6	6	6	6			6	6	6	6	6	6	6	6	6
0	1	2	3	4			6	7	8	9	a	b	c	d	e
x	x	x	x	x			x	x	x	x	x	x	x	x	x
6	6	6	6	6	6	6	6	6	6		6	6	6	6	6
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
0	1	2	3	4	5	6	7	8	9		b	c	d	e	f
x	x	x	x	x	x	x	x	x	x		x	x	x	x	x
6		6	6	6	6	6	6	6	6	6	6	6	6	6	6
5		5	5	5	5	5	5	5	5	5	5	5	5	5	5
a		a	a	a	a	a	a	a	a	a	a	a	a	a	a
0		2	3	4	5	6	7	8	9	a	b	c	d	e	f
x		x	x	x	x	x	x	x	x	x	x	x	x	x	x

Figure 1. Routing table of a Pastry node with nodeId 65a1x, $b = 4$. Digits are in base 16, x represents an arbitrary suffix.

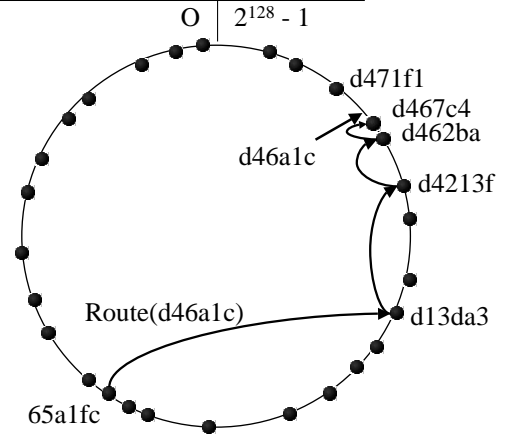


Figure 2. Routing a message from node 65a1fc with key d46a1c. The dots depict live nodes in Pastry's circular namespace.

randomly chosen contact node upon request. These techniques do not work well to find contact nodes for individual service overlays, which will likely be smaller and numerous. We describe a service that provides contact nodes for service overlays in Section 3.5.

3.2. Universal ring services

There are three basic services that the universal ring must provide to facilitate service advertisement, discovery and binding.

3.2.1 Persistent store

The first service is a persistent store for key-file pairs that provides efficient access to files given their keys. This service is used to store information about services, the code needed to run them, and lists of contact nodes for the different services. All stored files are immutable except contact lists, which do not require strong consistency semantics.

The functionality provided by the persistent store is similar to the one offered by PAST [13]. All files stored in the universal ring must be signed using a private key associated with a valid nodeId certificate.

A key-file pair is inserted in the store by using Pastry to route to the node in the universal ring whose nodeId is the numerically closest to the key. This node verifies the signature in the file and then replicates the file over the n numerically closest nodes in the ring to provide fault-tolerance against node failures.

The lookup of a file given a key also involves routing

a lookup request to the node in the universal ring whose nodeId is the numerically closest to the key. The node performing the lookup then receives a copy of the signed file, which it can verify.

Most files stored in the persistent store are small and can be aggressively cached. Both on insertion and lookup these files are cached on all nodes that participate in the routing. This caching (which was also used in PAST) is important to avoid overloading nodes when there are flash crowds; it prevents the nodes responsible for storing a file associated with a service from being overloaded if the popularity of the service increases dramatically in a short period of time. Code files can be large and in this case it might make sense to cache them less aggressively.

3.2.2 Multicast

The second basic service is an application-level multicast service, called Scribe [2, 14]. Nodes wishing to subscribe to a multicast group route a request to the node whose nodeId is numerically closest to the multicast groupId, called the group's root. Each node along the path of the request implicitly subscribes to the group, and adopts the previous node along the route as a child in the group's multicast tree. The request terminates when it arrives at the root, or at a node that is already subscribed to the group. Because membership management is distributed, the system is highly scalable.

A message is multicast to the group by sending it to the root. The root then forwards the message to all its children, and so on. When a topology-aware protocol like Pastry or Tapestry is used as the underlying p2p overlay, the result-

ing multicast trees have the property that nodes in successively smaller subtrees are increasingly near each other in the Internet. As a result, the multicast is very efficient, both with respect to delay and link stress [2].

Moreover, using a simple and efficient search algorithm, any node in the universal ring can efficiently locate a nearby member of a given group. To find such a member, a message is routed towards the group's root. When the message reaches a subscriber, it returns its list of children. The client then contacts the nearest among these children (determined, for instance, by measuring the RTT to each child). This process continues until a leaf in the multicast tree is reached. If the multicast tree was constructed in a topologically aware fashion, then that node is likely to be among the members that are nearest to the client issuing the search. Accuracy of this search can be traded for even higher efficiency by contacting a random child in each step; this works particularly well when members exist that are very close to the client.

The nearby subscriber search can be used to discover a nearby node in the network with a given property. Nodes with a given property subscribe to a group associated with the property. It can be used, for instance, to efficiently locate nearby nodes with certain hardware capabilities or services, nodes that have spare capacity, nodes that provide a specific service, or nodes that are operated by an organization trusted by the user.

3.2.3 Distributed search

The third basic service is a distributed search engine that allows users to find services given textual queries. Given a set of keywords and a service key, it associates the keywords with the specified key. The search engine allows nodes to search for keys using a set of query keywords. In the simple case, a boolean AND query is supported. More complex queries and ranking of query results are possible but details are beyond the scope of this paper.

We now briefly outline how the indexing could work. There are currently several projects looking at the development of searching and indexing for DHTs [5]. Here we describe a very simple scheme that can be significantly improved. The searching can be achieved by using a distributed inverted index that associates a keyword with a list of service keys. Every node in the universal ring stores part of the inverted index. The index for a keyword is stored in the node whose nodeId is numerically closest to the hash of the key. For resilience to node failure, the index on each node is replicated over the n numerically closest nodes in the ring. When a search is performed, the keywords in the query are hashed and Pastry is used to access the corresponding indices. If a keyword has an inverted index entry, the associated set of service keys is returned. The

node can then take the intersection of all the sets of keys returned. The intersection represents that set of services that satisfy the query. We plan to cache results of popular queries in the path to their component keywords to prevent overloads under flash crowds as was done in the persistent store.

Persistent queries (also called triggers) can be implemented as follows. A node that issues a persistent query subscribes to a Scribe multicast group associated with each keyword that appears in the query. When a service is advertised, a notification is sent on the multicast groups associated with each of the service's keywords. The receivers intersect the notifications received on each group to which they subscribe according to the query. As an optimization, boolean AND queries can be handled by subscribing to a group associated with the conjunction of query keywords in a canonical form. The root of such a group in turn subscribes to the groups associated with each of the conjunctive term's keywords and intersects notifications in the obvious way.

In the following sections, we describe in more detail how the persistent store, the multicast service and the search engine are used to enable discovery of services, code, and contact nodes.

3.3. Service advertisement and discovery

A service is created by generating a *service certificate* that describes the service. This certificate includes the textual name of the service, a textual description of the service, and a set of *code keys* (which are described in the next section). Each code key identifies a different implementation that provides the functionality required to run the service. The service certificate is signed by the private key associated with the nodeId certificate of its creator.

To advertise a service, the creator uses the persistent store provided by the universal ring to store the service certificate reliably under a *service key*, which is equal to the hash of the certificate. The textual description of the service and the service name are then inserted by the creator into the indices of the search engine provided by the universal ring. This associates the keywords with the service key.

In order for a node to retrieve the service certificate, it must discover the service key. This is performed by keyword searching using the search engine provided by the universal ring. A user performs a keyword search to retrieve a set of service keys, and then these service keys can be used to retrieve their associated service certificates from the persistent store provided by the universal ring. Alternatively, a node interested in certain categories of new services can issue a persistent query in the search engine, in order to be notified when new services of interest are ad-

vertised.

3.4. Code binding and update

As discussed above, we allow the creator of a service to specify several acceptable implementations for the service. These implementations are not necessarily written by the service creator and they may be used by many services that provide similar functionality. Therefore, code is stored separately from service certificates.

Each implementation has a *code certificate* that includes the implementation name, a textual description of the code, and the actual code¹. The certificate is signed by the code writer using the private key associated with its nodeId certificate in the universal ring. This signature allows users to verify that the code was written by the code writer, which is important because the user may be unwilling to run a piece of code just because the service creator vouched that it was suitable for its service.

The *code key* associated with the code certificate is obtained by hashing its contents. The persistent store provided by the universal ring is used to store the code certificate reliably under its code key.

After obtaining a service certificate, a node selects a code key, and then retrieves the code certificate associated with that key from the persistent storage service running on the universal ring.

Software updates for an implementation are inserted into the persistent store. The new code keys are then advertised on a multicast group consisting either of all members of the associated service overlay, or all nodes that use previous versions of the given implementation.

3.5. Joining a service overlay

After obtaining the service certificate and the code for a service of interest, a node is almost ready to join the service overlay. But first it needs to obtain the address of a contact node in the service overlay. We describe how to find this node next.

For each service, a small list of contact nodes is inserted in the universal ring under the service key. A node that wants to join the overlay of the service obtains this list when it looks up the service certificate in the universal ring. Then, it selects one of the nodes in the list at random to be its contact node.

To ensure that the contact list remains fresh, the oldest element in the list is replaced by the joining node. Copies of the contact list can be cached in the universal ring path

to the node that stores the service key to prevent overloading this node. Additionally, each cached copy of the list can be updated independently, as described above, to ensure its freshness and to prevent overloading of the contact nodes.

P2p overlays like Pastry [12] and Tapestry [17] exploit network locality to provide better performance. They require that the contact node be close to the joining node in the underlying network topology in order to achieve this. However, because of the randomization of nodeIds it is highly likely that the contact node is not close to the joining node. This problem can be solved by performing a nearest subscriber search on a multicast group consisting of the service overlay's current members.

Alternatively, in Pastry, the problem can be solved by using the algorithm described in [1]. This algorithm uses the contact node and traverses the service overlay routing tables bottom up to find a good approximation to the service overlay node that is closest to the joining node in the network. A similar algorithm could be used with Tapestry. Once the closest node has been found, it is used to start the joining algorithm described in [1].

4. Conclusions

In this position paper, we have outlined a preliminary design of an infrastructure that provides service advertisement, discovery and binding to bootstrap services based on structured p2p overlays. This problem has not been addressed by previous work.

We have proposed the use of a universal ring that provides only bootstrap functionality while each service runs in a separate p2p overlay. The universal ring provides: an indexing service that enables users to find services of interest by supplying boolean queries; a multicast service used to distribute software updates and for coordination among members of a service overlay; a persistent store and distribution network that allows users to obtain the code needed to participate in a service's overlay; and a service to provide users with a contact node to join a service overlay. These services are self-organizing and fault-tolerant and scale to large numbers of nodes.

The solution we have proposed, whilst targeted at Pastry, is applicable to other protocols such as CAN, Chord and Tapestry. It is also applicable to service discovery and binding for traditional centralized services.

References

- [1] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks, 2002. Submitted for publication.

¹Potentially other fields could be added to code certificate, such as a documentation URL, version number, code dependency information and so forth.

- [2] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002. To appear.
- [3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *18th ACM Symposium on Operating Systems Principles*, Oct. 2001.
- [4] J. R. Douceur. The sybil attack. In *Proceedings of IPTPS02*, Cambridge, USA, March 2002. <http://www.cs.rice.edu/Conferences/IPTPS02/>.
- [5] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *Proceedings of IPTPS02*, Cambridge, USA, March 2002. <http://www.cs.rice.edu/Conferences/IPTPS02/>.
- [6] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A scalable peer-to-peer web cache. In *Proceedings of the 21st Symposium on Principles of Distributed Computing (PODC 2002)*, Monterrey, CA, July 2002.
- [7] J. Kubiawicz and et al. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, November 2000.
- [8] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02*, Cambridge, USA, March 2002. <http://www.cs.rice.edu/Conferences/IPTPS02/>.
- [9] Microsoft. Upnp specification.
- [10] OMG. Corba naming service specification.
- [11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM*, Aug. 2001.
- [12] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001.
- [13] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles*, Oct. 2001.
- [14] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Third International Workshop on Networked Group Communications*, Nov. 2001.
- [15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.
- [16] Sun. Jini specification.
- [17] B. Y. Zhao, J. D. Kubiawicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, April 2001.
- [18] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiawicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proc. of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, June 2001.

Extended Abstracts

1. A Design of the Persistent Operating System with Non-volatile Memory
Author(s): Ren Ohmura, Nobuyuki Yamasaki, Yuichiro Anzai
Page: 148
2. AIMS: Robustness Through Sensible Introspection
Author(s): Febian E. Bustamante, Christian Poellabauer, Karsten Schwan
Page: 153
3. An Approach for a Dependable Java Embedded Environment
Author(s): Gilbert Caballic, Salam Majoul, Jean-Philippe Lesot, Michel Banatre
Page: 157
4. An Online Evolutionary Approach to Developing Internet Services
Author(s): Mike Y. Chen, Emre Kiciman, Eric Brewer
Page: 161
5. Applying source-code verification to a microkernel The Viasco project
Author(s): Michael Howmuth, Hendrik Tews, Shane G. Stephens
Page: 165
6. Back to the Future: dependable computing = dependable services
Author(s): Jeffrey Chase, Amin Vahdat, John Wilkes
Page: 170
7. Dependency on O.S. In long-term programs: Experience report in space programs
Author(s): Patrick Cormery, Le Vinh Quy Ribal, Arnaud Stransky
Page: 174
8. Design and Implementation of the Lambda u-Kernel based Operating System for Embedded Systems
Author(s): Kenji Hisazumi, Teruaki Kitasuka, Tsuneo Nakanishi, Akira Fukuda
Page: 178
9. Efficient Heartbeats and Repair of Softstate in Distributed Hash Table Systems
Author(s): Hakim Weatherspoon and John D. Kubiatowicz
Page: 182

10. Event-driven Programming for Robust software
 Author(s): Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazieres, Robert Morris
 Page: 186
11. Execution Time Limitation of Interrupt Handlers in a Java Operating System
 Author(s): Meik Felser, Michael Golm, Christian Wawersich, Juergen Kleinoeder
 Page: 190
12. Extensible distributed Operating System for reliable control systems
 Author(s): Katsumi Maruyama, Kazuya Kadama, Soichiro Hidaka, Hiro-tatsu Hashizume
 Page: 194
13. Fault Tolerance in Biomedical Systems
 Author(s): Shane Stephens
 Page: 198
14. Gaining and Maintaining Confidence in Operating Systems Security
 Author(s): Trent Jaeger, Antony Edwards, Xiaolan Zhang
 Page: 201
15. High-Confidence Operatins Systems
 Author(s): Radu Grosu, Erez Zadok, Scott A. Smolka, Rance Cleaveland, and Yanhong A. Liu
 Page: 205
16. Increasing smart card dependability
 Author(s): Ludovic Casset, Jean-Louis Lanet
 Page: 209
17. Making Sound Tradeoffs in State Management
 Author(s): George Candea, Armando Fox
 Page: 213
18. Model Checking System Software with CMC
 Author(s): Madanlal Musuvathi, Andy Chou, David Dill, Dawson Engler
 Page: 219
19. OASIS project: deterministic real-time for safety critical embedded systems
 Author(s): Stephane Louise, Vincent David, Jean Delcoigne, Christophe Aussagues
 Page: 223

20. Operating System Support for Massive Replication
Author(s): Arun Venkataramani, Ravindranath Kokku, Mike Dahlin
Page: 227
21. Pangaea: a symbolic wide-area file system
Author(s): Yasushi Saito, Christos Karamanolis
Page: 231
22. Replica Management Should Be A Game
Author(s): Dennis Geels, John Kubiawicz
Page: 235
23. Secure Coprocessor-based Intrusion Detection
Author(s): Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ron Perez, Reiner Sailer
Page: 239
24. THINK: a secure distributed systems architecture
Author(s): Christophe Rippert, Jean-Bernards Stefani
Page: 243
25. Time Fault Removal for Safety-Critical Real-Time Embedded Systems
Author(s): Sebastien Faucou, Anne-Marie Deplanche, Yvon Trinquet
Page: 247
26. Towards Trusted Systems from the Ground Up
Author(s): Vivek Haldar, Michael Franz
Page: 251
27. Why do internet services fail and what can be done about it?
Author(s):
David Oppenheimer, David A. Patterson
Page: 255

A Design of the Persistent Operating System with Non-volatile Memory

Ren Ohmura

Nobuyuki Yamasaki

Yuichiro Anzai

Graduate School of

Science for Open and Environmental Systems

Keio University, Japan

{ren,yamasaki,anzai}@ayu.ics.keio.ac.jp

Abstract

In today's computing environment, novel memory devices with non-volatile characteristics are increasing in practicality when used as the main memory, due to the persistence with no additional battery that significantly enhances usability of personal devices.

In our research, we have built a persistent operating system using non-volatile main memory. This paper describes our strategy in detail on how atomicity of execution is maintained for each device driver method so that the state of peripheral devices can also be recovered consistently. The method was implemented on the Linux kernel using a UART device driver. We have confirmed correct system recovery through our experiments.

devices". It was thus difficult to preserve the system state in a short period without decreasing system performance.

Recently, memory devices with non-volatile characteristic that can operate without any batteries (e.g. FeRAM and MRAM) are becoming more practical. As these memory devices increase in their capacity and speed, they can be used as the main memory.

Therefore, we are currently developing a persistent operating system on a non-volatile main memory system towards fine grain persistence without any additional battery. Even on such a system, not only does the main memory state but also the CPU and peripheral device states must be recovered consistently after power failure. This paper describes our basic concept on how to recover the consistent system state while ensuring atomicity of the device driver method execution. Our experiment was implemented the Linux kernel.

1 Introduction

Many computer systems today lose their application context when they are faced with an unpredictable power failure. Additional power supplies (e.g. UPS or second battery) are needed to protect the system against this problem. However, this increases the size and cost of the system especially in personal devices such as PDAs or Set-Top-Boxes.

Past experiments on *persistent operating systems* make the application's execution states and data recoverable significantly increasing the reliability and usefulness of computer systems. Based on this system, a user can continue his/her task after an unpredictable power failure with minimum data loss without rebooting the OS and re-executing each application, which takes enough time to interrupt the user's work. Furthermore, if persistence is achieved in a pretty short periods, a computer system can be made to run with an unstable power source such as solar power. However, past experiments have assumed that permanent devices in the system are either disks or tapes, known to be "slow

2 Motivation

Most existing research on the persistent operating system focus on when and how to store the main memory state with the CPU state into the permanent storage devices such as disks[3, 4, 5, 7].

On a system with non-volatile main memory, the CPU automatically saves the current main memory state from each store instruction. The CPU state can be saved in a similar way to the context switch, by writing to the main memory. We developed a method based on this to save both states consistently with low overhead using explicit timing called "checkpoints". It copies the memory space modified since the previous checkpoint to free space, similar to the "side file" scheme, while arranging the memory management structure to reduce overhead. Unfortunately, in the worst-case scenario, this would require twice as much as the normal memory space and needs time to save all the changed memory space and the CPU state, which is unavoidable according to the side file scheme. Even if logging

scheme is used, the overhead increases even more because logs have to be taken on each instruction.

Few experiments examine the peripheral devices state. In order to continue system execution, the all states of the main memory, CPU and peripheral devices must be recovered because all are intertwined with each other.

Thus, our goals are:

- to be able to save and restore the entire consistent state of the system including peripheral devices
- to reduce memory space and the time required to store system states

To achieve these goals, we focused on controlling the execution of device drivers since there are power management schemes that make suspend/resume and hibernation possible, such as APM and ACPI [2, 1]. However, they are unable to handle an unpredictable power failure because they save the state of peripheral devices during special time when they can.

3 Design

The type of CPU determines the amount of the CPU state that needs to be saved. The system hardware can be designed to allow sufficient time to store the state during power failure, which can be detected by an interrupt in a circuit monitoring the power level, by slowing the power attenuation with the device like capacitors. If the system consisted of only the (non-volatile) main memory and CPU registers (used for calculation), it would recover the execution correctly with the CPU state stored during power failure and the memory state existing on the main memory. Hence, only the CPU state is stored when a power failure occurs. Existing memory state, except the devices driver areas, is used for recovery, which reduces overhead.

However, the amount of peripheral device states required to be saved cannot be estimated at system hardware design time. Most systems allow users to add peripheral devices to extend the system (e.g. PCMCIA cards). Some states of devices cannot be read by the CPU, which makes it impossible to store all of the states of peripheral devices during power failure. Furthermore, many devices require some access to begin and perform asynchronously, so they may act incorrectly or harm the system in the worst case due to the loss of the previous access and states of peripheral devices if the system restarts from the precise point of power failure.

Therefore, callback functions written in device drivers like APM and ACPI are used for recovering the states of devices at that explicit point. We defined that point as the head of each device driver method. The state of the CPU and the memory space used by a device driver are saved at the beginning of each device driver method. If the CPU context is in a device driver when a power failure occurs, the recovery operation restores this memory state, calls the

callback function to recover the consistent device state, and then restarts the system with the saved CPU state. This way, the system restarts from the head of the device driver method, and repeat it with the consistent device state as each unit of request for the devices.

In essence, our basic concept for maintaining consistency of the entire system state is to ensure atomicity of execution of each device driver method. In the following sections, we will describe the Linux device driver, the implementation target, and discuss in detail the design of our strategy.

3.1 Linux device driver

Although there are many peripheral devices and methods to implement them, we observed a more simple case. Each peripheral device is managed as a file in Linux distinguished by a `kdev_t` value, which is a set of major and minor numbers¹. When a user application calls a device handling function, such as `open`, `read`, `write`, or `close`, the file operation handler looks up the appropriate device driver and calls a suitable method in it.

Regular kernel functions used in device drivers are generally fixed. For example, `wakeup` and `down` and other similar functions are used to wake up and put threads to sleep; `kill_fasync` is used for handling asynchronous device signals; and `queue_task` is used for queuing the thread of other kernel components called the *bottom half handler* or *tasklet*, such as protocol stacks.

3.2 Basic Operations

Figure 1 shows the basic sequence of the file operation handler extended by our scheme while entering into and returning from a device driver method. When a user issues a request to a device, shown in step (1) in Figure 1, the file operation handler looks up the appropriate device driver. Before calling the method, it saves the memory of this device driver and the CPU states to the area specified by `kdev_t` value, shown in step (2) and (3). Then, it calls the method in step (4). After returning from the device driver, it destroys the CPU state saved in step (3), shown in step (5), and returns to the user level shown in step (6).

The recovery process looks for the CPU state saved in (3) at first. Next, if there is no CPU state relating with any device driver, the recovery process restarts the system with the CPU state saved during power failure. If there is one or more CPU states, the recovery operation restores the device driver's memory state saved at (2) and then restarts the system with the CPU states saved at (3).

Nevertheless, we have to consider following conditions:

¹Although network devices are not managed as a file, we leave this matter to next paper.

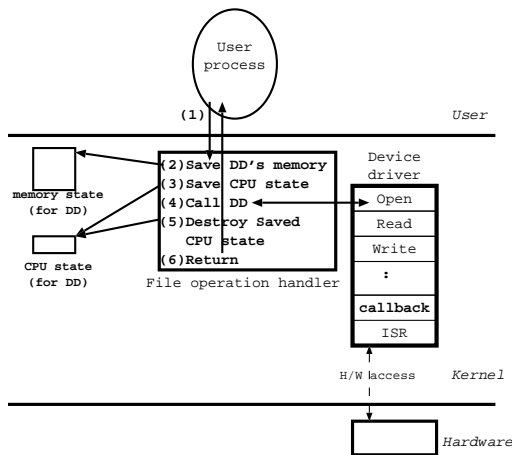


Figure 1. The operations in the file handler

context switch in a device driver
cooperation with external components
interrupt handling

3.3 Context switch

There are many cases that a context switch occurs in the device driver due to the lack of required data, mutual exclusion, and so on.

If a thread context switches in a device driver and a power failure occurs while another thread is running outside, the CPU state saved during power failure is that of the outside thread. In order to restart the thread inside the device driver from the start of that method, the recovery process overwrites the CPU state of this thread on the context table with that saved at step (3) in Figure 1. The subsequent scheduling point of this thread is changed from the inside to the head of the device driver by this operation.

In addition, the state of this thread (e.g. *running* or *blocked*) has to be recovered. Therefore, the thread state is also recorded by step (3), and the recovery process overwrites it on the context table.

3.4 Cooperation with external components

Generally, device drivers cooperate at the user level or with other kernel components, writing data to or reading data from the buffers. These buffers are not required to be restored during the recovery process.

If the device driver method reads from a buffer, the current data at the buffer is the same before power failure. Then, the operations of the method can re-execute with the same data after recovery. When writing to a buffer, the recovered request executes the same operations and gets the

same or more appropriate data. Then external components work correctly. Thus, the buffer concerning external components does not need to save and recover. This observation decreases the overhead of our scheme.

However, pointers and index variables of the buffers need to be recovered as indicating the same position of the buffer after recovery.

3.5 Interrupt

When a power failure occurs while handling an interrupt, the interrupt acts as if it had not been executed.

Interrupts from peripheral devices can be roughly classified into two groups. One is the result of a request from the device driver. The other is an active event from a device such as a button press or a network packet arrival. In the first case, the same interrupt is expected by repeated requests. In the second case, it does not matter since the interrupt acts as if it had never occurred.

A typical interrupt service routine does one or a combination of the following operations with the kernel functions mentioned in section 3.1. The routine registers a request for the execution of other kernel components; it sends a signal to the corresponding thread as notification of asynchronous I/O; and it wakes the sleeping thread waiting for an event (data) from a device.

If a power failure occurs after they are operated, the recovery process invalidates the operations. For this reason, kernel functions, such as `queue_task`, `kill_fasync` and `wakeup`, record these requests as a log.

4 Implementation

4.1 Kernel Function

As noted in section 3.4, it does not need to save all of the memory in a device driver. To reduce overhead and maintain flexibility of device driver implementation, we added a new kernel function that registers the memory space to save when entering the device driver. The arguments in this function are the `kdev_t`, the head address of the memory, and the size. The initialization code in a device driver, which is only executed during the first boot sequence, registers the memory space using this function.

We also extended some kernel functions, such as `queue_task`, `fasync_kill`, `wakeup`, et cetera, as mentioned above.

4.2 Callback Function

The difference between the callback functions in our scheme and the APM and ACPI is that they have the responsibility to recover complete states of peripheral devices

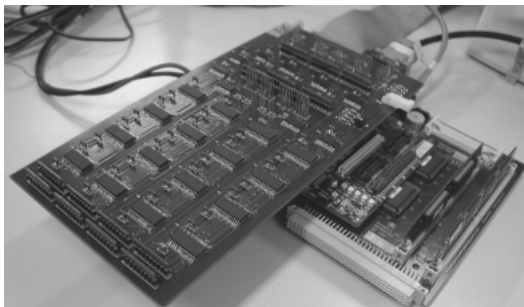


Figure 2. FeRAM board and MPC860FADS

at the head of the device driver method. In order to do that, the device driver has to be written as storing the each change of the device state to the registered main memory and callback functions reflect it after initializing the device.

For example, with the UART device, current configurations, such as the baud rate, the number of stop bits, and the number of parity bits, are kept in the registered main memory. The callback function, called after restoring the memory state, has to set these to the device as the current (at the head of the method) state.

4.3 Recovery Operation

The steps for recovery operation are:

1. Recover the memory state of device drivers where the saved CPU contexts exist
2. Call the callback functions of device drivers
3. Overwrite appropriate contexts on the context table with the CPU state at the beginning of the device driver method (section 3.3)
4. Change thread states which had their context switched in device drivers (section 3.3)
5. Undo requests from the device driver based on logs(section 3.5)

Then, if the CPU state saved during power failure is neither in the device driver nor in the interrupt handler, the system restarts with this CPU state. Otherwise, if it is in the device driver, the system restarts with the same thread in the CPU state, saved at the head of the device driver method. If it is in the interrupt handler, the scheduler restarts.

5 Experimentation

We tested our strategy on the MPC860FADS, an evaluation board of the PowerPC core CPU for embedded system, with FeRAM boards as the main memory (Figure 2). The prototype was implemented on Linux 2.4.4 with a UART device driver.

We ran a simple process, which incremented a counter value and printed it on the UART, and then, shut off the power supply. We confirmed that the system showed the next value before power failure on the UART and continued performing correctly. At that time, the registered memory space was only 152 bytes and the increased execution time ratio measured by the `time` command was less than 0.5%.

6 Summary and Future Work

We illustrated our strategy of recovering an entire system state consistently given the condition that the main memory is non-volatile, focusing on the device driver. The basic concept of our scheme is to ensure atomicity of the device driver method, observing whether there is thread present in the device driver at power failure. We implemented our prototype on the Linux kernel and its UART device driver, and confirmed the system including the actual device performed correctly after recovery.

The experimentation we conducted was only a simple case, assuming that only one thread is in the device driver. However, more than one thread usually enters the same device driver simultaneously, especially in sharable devices. Also, we did not consider some device driver method such as `mmap`. We plan to solve these questions towards the computer system performing with unstable power supply.

References

- [1] *Advanced Configuration and Power Interface Specification Revision 2.0*, July 2000. <http://acpi.info/spec.htm>.
- [2] *Advanced Power Management BIOS Interface Specification Revision 1.2*, February 1996. http://www.microsoft.com/hwdev/archive/BUS-BIOS/amp_12.asp.
- [3] A. Dearle and D. Hulse. Operating system support for persistent systems: past, present and future. *Software – Practice-and-Experience*, 30(4):295–324, 2000.
- [4] K. Elphinstone, S. Russell, G. Heiser, and J. Liedtke. Supporting Persistent Object Systems in a Single Address Space. In *Proc.7th POS*, 1996. <http://www.cse.unsw.edu.au/~disy/papers/index.html>.
- [5] A. Lindstrom, A. Dearle, R. di Bona, S. Norris, J. Rosenberg, and F. Vaughn. Persistence in Grasshopper Kernel. In *Proc. of the Eighteenth Australian Computer Science Conf.*, pages 329–338, 1995.
- [6] A. Rubini and J. Corbet. *Linux Device Drivers, 2nd Edition*. O'Reilly & Associates, Inc., June 2001.
- [7] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *17th ACM Symposium on Operating System Principles(SOSP '99)*, pages 170–185, 1999.

AIMS: Robustness Through Sensible Introspection

Fabián E. Bustamante, Christian Poellabauer and Karsten Schwan
 College of Computing, Georgia Institute of Technology
 Atlanta, Georgia 30332
 {fabianb, chris, schwan}@cc.gatech.edu

1 Introduction

As individuals and organizations increasingly rely on services provided by complex computing systems for their everyday tasks, system dependability becomes a main concern. The dependability of a system is commonly defined as the property of it such that reliance can justifiably be placed on the service it delivers [1] and includes, as special cases, attributes such as reliability, availability, safety, and security. In addition, dependable systems must also be robust when facing the “messiness” of the real world, delivering correct service across a wide range of operational conditions and failing gracefully outside that range [15, 16].

A common approach to obtain robust dependable systems is trying to predict the system’s future operational conditions and provision it accordingly. This approach, however, normally translates on high costs due to over-provisioning and costly system updates (for its web site, Charles Schwab maintains capacity at three to five times peak volumes [22]), and when predictions fail it can result on high average down-time with its associated costs¹, estimated at about \$8,000 per hour [5, 25].

In response, the systems community has identified as an important focus for research the design of computing systems capable of adapting to predictable changing environments [2, 11, 12, 20] and, in this context, introspection has proven to be a useful approach [3, 4, 13, 17, 18, 23]. Introspection is the ability to continuously monitor system behavior and adapt to changing conditions. Central to the process of providing introspection is the collection, aggregation, and processing of monitoring data. Probes are inserted in different parts of the systems to collect raw monitoring data about current hosts’ hardware capabilities, dynamically compute statistics on environmental or systems conditions, or collate information on systems requests. The collected data is then selected and integrated to produce system-specific metrics, diagnose potential problems, and

select among different adaptive measures to enact.

For robustness, however, the introspective component itself needs to be dynamically adaptive: the same complexity that initially drove us to adopt introspection along with the “messiness” of the real world mean that the parts monitored, the monitoring granularity and even the processing done on the collected data are bound to change dynamically. Since (i) it is effectively impossible to predict all information needed for introspection, (ii) even if we try, no introspective system will be able to manage the amount of data necessary to select the right adaptation to an overwhelming number of possible system conditions, for the introspective component to be useful in the real world it must be dynamically adaptive, and (iii) the “right” adaptation may be situation dependent as well.

Our work at Georgia Tech focuses on exploring the idea of dynamically adaptive introspective components for future systems. To this end, we are building *AIMS*, an *Adaptive Introspective Management System* through which monitoring probes (or *agents*) can be (un-)installed at runtime, their execution can be finely tuned dynamically, and the processing done on the collected data can be changed as needed.

In the reminder of this paper, we describe the proposed architecture, present some details on our current implementation, and describe our initial steps for the application of our ideas on a public *ftp* service.

2 Architecture

AIMS offers a push-based/active customizable service for environment- and self-monitoring. Hosts of interest to an application must become AIMS nodes. Different objects (including devices) at AIMS nodes are the sources of monitoring data. AIMS nodes connect to other nodes in specific sets or AIMS networks. AIMS clients connect to AIMS networks through an AIMS node that could reside on its own or another host. Clients express their interests in different monitored objects by requesting state reports on such objects or selectively registering themselves with the moni-

¹Delta Air Lines experiences recovery delays of up to several hours in response to partial system outages [10] and eBay’s 22-hour crash in June 1999 cost the company more than \$5 million in returned auction fees [25].

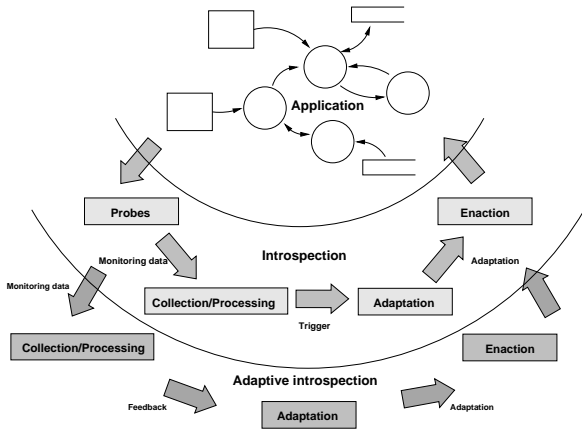


Figure 1. AIMS adaptive introspection.

toring streams associated with those same objects. Requests for continuous reports follow a *lease* model: they remain active for a user-specified (but limited) time span after which they are cancel. Previously issued requests can be re-issued or canceled, before lease expiration, using their request IDs.

Clients can select a subset of the data made available by different sensors, and integrate that data to produce application-specific monitoring metrics and decide on possible adaptations. In addition, filters in AIMS can be instantiated directly in the source of monitoring data streams with the corresponding savings in bandwidth, and in sink or source processing.

Sensors maintain histories of measurements. In order to forecast performance availability, AIMS includes a library with a number of predictive algorithms including mean-based methods such as running, trimmed, and sliding window averages as well as minimum, maximum, median and support to build autoregressive models.

An application using AIMS can easily redefine, at runtime, its monitoring views by adapting/replacing the filters used for monitoring as its workload and environment change. In this manner, applications have an additional level of adaptation that would not be possible were the monitoring and adaptation mechanisms coded statically for each of them.

Figure 1 depicts an adaptation hierarchy: distributed system resources are monitored and adapted (or controlled) by the first-level adaptation mechanisms. At the second level, the effectiveness of this adaptation is evaluated, and the first-level monitors and controllers are adapted, if required.

As the operational environment of an application changes and the administrator’s understanding evolves, the focuses of monitoring and possible or useful adaptations change as well. Researchers have developed a multiplicity of tools for system instrumentation even helping developers

and users select suitable instrumentation points. There is, however, the danger of over/under instrumentation. Overly aggressive instrumentation results in potentially overloaded and therefore, non-robust systems. Too conservative instrumentation may results in insufficient information to make sound adaptation decisions. On the other hand, wrong adaptation decisions (perhaps due to insufficient or “old” monitoring information) can be non-effective or directly counterproductive [13]. In sum, the monitoring, data processing, and adaptation part of the introspective component need to be dynamically adaptable. The monitoring data filters/analyses and the adaptation tasks performed based on them need to be (re-)deployed dynamically and executed efficiently; there is a need for light-weight ways to adapt sensors functionality, data processing and the adaptation routines themselves. The goal of adapting the introspection system is to improve the component’s efficiency for a given execution instance: tuning the granularity of sensors, selecting the “right” prediction utility for the given monitoring stream’s characteristics, reducing the monitoring overheads and perturbations [9, 24], choosing the “right” adaptation at a given point in time, and dealing with unpredictable disasters [17]. Clearly, all this needs to be done without making the adaptive introspective component so heavyweight as to deny its benefits.

3 Implementation Details

Currently, AIMS nodes report information on memory availability, CPU load, disk free space, up-time, and number of users currently logged on, as well as host name, IP, number of CPUs, and configure triplet (as reported by GNU config.guess). Status reports of network paths between two AIMS nodes include latency and bandwidth. AIMS uses a combination of passive and active sensors as needed [26]. Passive sensors exercise an external system utility, such as `uptime`, and scan the utility’s output to obtain the required information. Active sensors, on the other hand, must conduct a performance experiment, such as timing a message exchange between two hosts, to measure the availability of the monitored resources.

A recent result of our work includes dynamically insertable kernel agents, which monitor system resources. The feature of this mechanism distinguishing it from past work is its ability to customize the monitors according to a client’s specific needs. Applications manipulate these agents via an interface implemented as an extension to the Linux virtual file system `/proc` [14].

For adaptation, our approach includes the adaptation of kernel-level resources. While it is straightforward to inspect and manipulate system resources within the kernel, it takes costly interfaces to present to user-level manager (often repeated system calls are needed to determine sys-

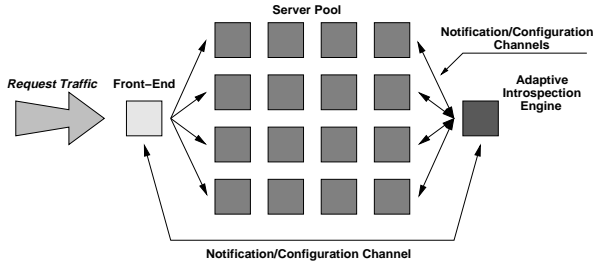


Figure 2. Cluster-based service with adaptive introspection.

tem information or adapt system resources). In addition, we have found that fine-grain kernel-level management of resources can provide applications with benefits not derived from coarser-grain, user-level management [13].

Resource controllers modify resource allocations to a specific application based on the requirements of the application and the availability of resources. Resource controllers use information from monitoring agents for their decision-making.

We have built a kernel-based event service aimed at supporting the coordination among multiple kernel services in distributed systems [19]. Event and publish/subscribe services have become prevalent in distributed applications ranging from virtual reality, avionics, to support for mobile users [8, 21, 6]. Our current work investigates the utility of a kernel-based event service for resource management, distributed monitoring, and load balancing mechanisms.

4 An Example in Cluster-based Servers

We have started to apply this work to the monitoring of cluster-based Internet services. In particular, we are working on an *ftp* service for publicly available software provided by the CERCS research center in the College of Computing at Georgia Tech. The high-level architecture of the adaptive Internet service (Figure 2) is similar to the one described by Chase et al. [7]. As with their work, the goals are to maximize server performance and minimize the cost (energy) by concentrating requests on a minimal set of servers. That is, some servers run at near 100% utilization, while others are idle.

A cluster of servers consists of a front-end node and a number of back-end nodes, with incoming requests being forwarded by the front-end to one of the back-end nodes. The individual cluster machines monitor their loads (CPU utilization, queue lengths, etc.) and exchange such information with each other via a kernel-level event channel. This information is used (i) as a basis for admission control and

(ii) for load balancing. Further, this information is also collected by a centralized **adaptive introspection engine**, which is able to adapt the front-end (e.g., to concentrate requests or to react to failing backends) and the back-ends (e.g., to vary queue lengths, change resource allocations, or modify forwarding algorithms). This forms the first level of the adaptive introspection mechanism. The second level is formed by monitors at the hosts measuring the effectiveness of the monitors and adaptation mechanisms. Again, this information is collected at the central adaptive introspection engine via another event channel. The engine is able to modify the behavior of the monitors (e.g., frequency of monitoring), the behavior of the event service (e.g., filters aggregate events or block events from being sent to idle machines), and the controllers (e.g., change thresholds or algorithms).

5 Conclusion

Our society increasingly relies on dependable complex computing systems. To be useful, dependable systems must also be robust when facing unpredictable changes to their operating environments. Introspection has proven to be a helpful approach in the design of dynamically adaptable computing systems. We argue that, for robustness, the introspective component itself needs to be dynamically adaptive since (i) it is effectively impossible to predict all information needed for introspection, (ii) even if we try, no introspective system will be able to manage the amount of data necessary to select the right adaptation to an overwhelming number of possible system conditions, and (iii) the “right” adaptation may be situation dependent as well.

At Georgia Tech we are exploring the idea of dynamically adaptive introspective components for future systems. To this end, we are building *AIMS*, an *Adaptive Introspective Management System* through which monitoring probes (or *agents*) can be (un-)installed at runtime, their execution can be finely tuned dynamically, and the processing done on the collected data can be changed as needed.

We are exploring our ideas on the context of a cluster-based *ftp* service, consisting of a front-end and a number of back-ends and a centralized adaptive introspection engine. Our future work will use adaptive mechanisms in the front-end and in the back-ends to adjust admission control and load balancing mechanism. AIMS will adapt these adaptation mechanisms to maximize the performance and robustness of the introspective system.

References

- [1] Mario Barbacci, Mark H. Klein, Thomas H. Longstaff, and Charles B. Weinstock. Quality attributes. Technical Report

- CMU/SEI-95-TR-021, Carnegie Mellon University, Pittsburgh, PA, 1995.
- [2] Thomas Bihari and Karsten Schwan. Dynamic adaption of real-time software. *ACM Transactions on Computer Systems*, May 1991.
 - [3] Aaron Brown, David Oppenheimer, Kimberly Keeton, Randi Thomas, John Kubiawicz, and David A. Patterson. Isstore: Introspective storage for data-intensive network services, March 1999.
 - [4] Fabián E. Bustamante, Greg Eisenhauer, Patrick Widener, Karsten Schwan, and Calton Pu. Active streams: An approach to adaptive distributed systems, May 2001.
 - [5] Bruce Caldwell. Cost of downtime. *Information Week*, May 12 1997.
 - [6] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.
 - [7] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, and Amin M. Vahdat. Managing energy and server resources in hosting centers, October 2001.
 - [8] Greg Eisenhauer, Fabian E. Bustamante, and Karsten Schwan. Event services for high performance computing. In *Proc. of the 9th High Performance Distributed Computing (HPDC-9)*, Pittsburgh, PA, August 2000.
 - [9] Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. Falcon - toward interactive parallel programs: The on-line steering of a molecular dynamics application, August 1994.
 - [10] Ada Gavriloska, Karsten Schwan, and Van Oleson. Adaptable mirroring in cluster servers, August 2001.
 - [11] Steven D. Gribble. Robustness in complex systems, May 2001.
 - [12] IFIP WG10.4 and IEEE-CS. [Dependability.org](http://www.dependability.org). www.dependability.org.
 - [13] Daniela Ivan-Rosu, Karsten Schwan, and Sudhakar Yalamanchili Rakesh Jha. On adaptive resource allocation for complex, real-time applications, December 1997.
 - [14] Jasmina Jancic, Christian Poellabauer, Karsten Schwan, Matthew Wolf, and Neil Bright. dproc - extensible run-time resources monitoring for cluster applications, April 2002.
 - [15] Philip Koopman and Henrique Madeira. Dependability benchmarking & prediction: a grand challenge technology problem, November 1999.
 - [16] J.C. Laprie, editor. *Dependability: Basic concepts and terminology in English, French, German, Italian and Japanese*. Springer-Verlag, 1992.
 - [17] Jeffrey C. Mogul. Operating systems support for busy Internet services, May 1995.
 - [18] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility, October 1997.
 - [19] Christian Poellabauer, Karsten Schwan, Greg Eisenhauer, and Jiantao Kong. Kecho - event communication for distributed kernel services, April 2002.
 - [20] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base: using abstraction to improve fault tolerance, December 2001.
 - [21] Antony Rowstron, Anne-Marie Kermarrec, Peter Druschel, and Miguel Castro. Scribe: The design of a large-scale event notification infrastructure. In *Proc. of the 3rd International Workshop on Networked Group Communication (NGC 2001)*, UCL, London.
 - [22] D. Scott. Web site availability and scalability: expert user advice. *Gartner Group Research Notes*, December 2000.
 - [23] Margo I. Seltzer and Christopher Small. Self-monitoring and self-adapting systems, March 1999.
 - [24] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels, February 1999.
 - [25] Tim Wilson. The cost of downtime. *Internet Week*, July 30 1999.
 - [26] Rich Wolski, Neil Spring, and Chris Peterson. Implementing a performance forecasting system for metacomputing: The Network Weather Service, November 1997.

An Approach for a Dependable Java Embedded Environment

Gilbert Cabillic

Salam Majoul

Jean-Philippe Lesot

Michel Banâtre

IRISA

Campus Universitaire de Beaulieu

35042 Rennes Cedex

Gilbert.Cabillic@irisa.fr

Abstract

A Java Execution Environment (JEE) presents lots of advantages for embedded architectures. In this paper, we present an approach to build a dependable Java execution environment for Wireless PDAs. It is based on a modular software architecture. Our on-going works focus on reducing software errors, increasing the security of the software and minimizing native software pieces of code to increase the overall stability of the platform by transferring operating system features in Java world.

1. Introduction

A Java Execution Environment (JEE) presents several advantages for embedded architectures. First, Java applications can be dynamically downloaded. Second, as Java bytecode is an intermediate code, the application can be distributed everywhere. Moreover, a Java application cannot explicitly manage memory access because Java bytecode provides no way to express and manipulate pointers in Java world. This increases the stability of all the Java world by avoiding memory access errors due to software faults or illegal intrusion. Of course, this stability depends on the JEE (Java virtual machine (JVM), and APIs implementation), and also on the underlying operating system.

The software architecture we consider is presented in figure 1. Our Java platform (Java virtual machine and Java APIs), named Scratchy [3], runs on the hardware through an operating system. Scratchy is executed in several OS threads in one common space address. Scratchy has the ability to execute several applications at the same time and a scheduler inside Scratchy undertakes the scheduling of all Java threads. Native code, independent of Java world like protocol layers or driver daemons, is executed in other OS threads thanks to the help of the operating system.

In this paper, we present an approach to build a dependable Java execution environment for Wireless PDAs. It is

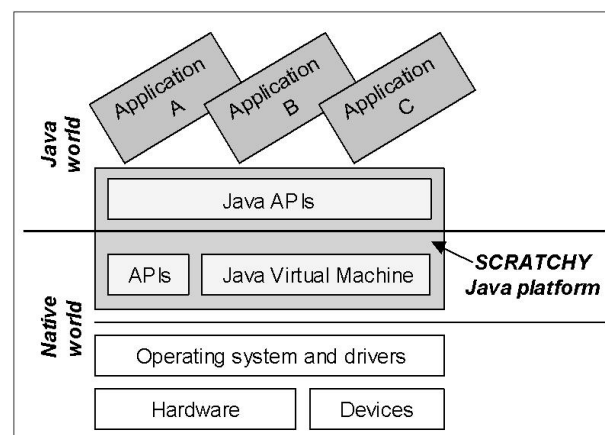


Figure 1. Global software architecture.

organized as follows. First, we present our Scratchy modular software architecture. Then, features of our development environment for Scratchy are mentioned as well as some evaluations. Finally, we present our perspective works to increase the dependability of our Java execution environment.

2. Scratchy Software Architecture

Memory volume, execution time and energy consumption are critical resources for embedded systems. A flexible Java environment providing a good tradeoff between these resources is required for wireless PDA. Only a modular approach, whose benefits have been already proven in software design, allows to achieve such a tradeoff. In fact, through modularity it is possible to specialize some parts of the JVM for a specific processor by exploiting, for instance, low power features for DSP. With a monolithic-programmed Java environment, it is difficult to change at many levels hardware resource management without rewriting the JVM from scratch for each platform.

2.1 Our modular approach

To reach these goals we designed a modular approach described in the following. A module contains services (functions) and data type structures. Pre-hooks and post-hooks can be attached to a service. The formers are called before a service call to undertake for example resource reservation and allocation. Post-hooks are called after a service call for instance to free the resources and manage errors.

The Scratchy Development Environment (SDE), presented below, is designed to achieve modularity for our Java environment.

2.2 Scratchy Development Environment

This tool is designed to optimize time or memory overhead on the outcome source and to be the most language and compiler independent as possible. SDE takes four inputs:

- a global specification file describes services and data types by using an Interface Definition Language (IDL);
- a set of modules implements services and data types in one of supported language mappings;
- an implementation file inside each module describes the link between specification and implementation;
- target hardware descriptions indicate alignment constraints with their respective access cost for each target processor.

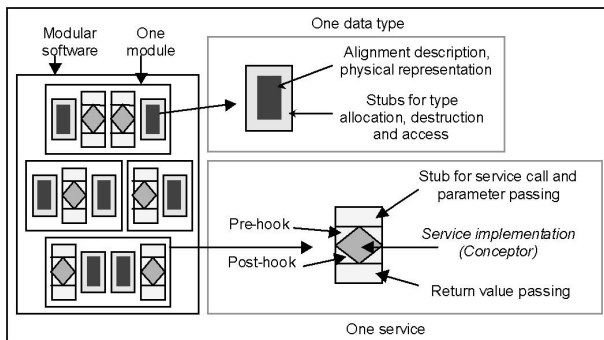


Figure 2. Our modular approach.

SDE chooses a subset of modules according to the targeted hardware criteria. It generates, as shown in figure 2, stubs for services, sets structures of data types, generates functions to dynamically allocate and access data types, etc. SDE works at source level, so it is the responsibility of compilers or preprocessors (through in-lining for example) to optimize a source which has potentially no overhead.

2.3 Some evaluations

The current version of SDE supports the C language as language mapping and generates stubs for services and data type structures in that language.

In order to evaluate memory expansion of SDE data type structures, their memory allocation time and service execution time, we compared the generated stubs to an equivalent program written in C. For this purpose, we compared the assembler codes obtained after optimization for both programs. We used benchmarks when the assembler codes were not comparable. The results obtained were very promising as the generated stubs introduce an insignificant overhead and in some cases no-overhead at all.

For example, the assembler codes of a C program calling the generated stub for a service without parameter and the C program calling directly the C function implementing that service are identical. When the service is specified with parameters, the assembler code for the SDE program contains only one supplementary statement relating to a shift of the stack pointer.

We made some benchmarks to evaluate data type structures generated by SDE. We used a Pentium II processor under Linux system and using a Gnu C compiler and optimizer. As an example, figure 3 shows the time required to allocate an array generated by SDE and an equivalent C array. The time measured to access an array element is the same in both cases.

Array size	C Program (cycles)	Using SDE (cycles)	Overhead (cycles)
20	229	228	+1
80	228	228	0
280	228	228	0
400	228	228	0

Figure 3. Array allocation time comparison.

2.4 Related works

Our modular approach is close to aspect-oriented programming (AOP) that relies on a separation phase of different aspects of a problem and a composition phase. AspectJ [5] is an example of AOP language. It provides some constructs similar to that proposed by SDE such as pre and post-conditions that act like our pre and post-hooks. Another tool that can be compared to SDE is Knit [8] which provides its own language to describe linking requirements. However, both AspectJ and Knit are too limited for our goals because they manage only method/function calls, and not data type. Management of data type is useful to optimize memory consumption and data access performance especially on shared memory multiprocessor hardware.

2.5 Implementation

Scratchy is our modular implementation of the JVM which is written from scratch in order to validate our modular approach. It relies on a CLDC [10] specification compliant. However, Scratchy is closer to CDC than to CLDC because it supports floating-point operations and Java Native Interface. Pentium, ARM and a TI DSP TMS320C55x [11] targets are supported by our JVM as well as a bi-processor Pentium based architecture as mentioned in [3]. To make the link with an operating system, we designed a module named middleware which supports not only a Posix general operating system (Linux, Windows 2000) but also a real-time operating system with Posix compatibility (VxWorks).

3 Perspective works

The stability of the overall platform depends on 3 different points. First, Java applications may contain software errors either deliberately introduced in case of intrusion, or not in case of erroneous code. As all explicit memory manipulations are forbidden, only the involved application is affected when a software error occurs. A software exception (Java language) can be sent and in the worst case, when the error cannot be recovered, the application is killed by the JEE. A Java application can disrupt the global software architecture when its requirements in terms of resources are too important (memory, CPU, network) or when the application uses an API containing a potential software error that could damage devices or lead to a memory error access. Second, the JEE itself can contains software errors in Java or native code part. Third, the operating system and drivers can be instable.

In order to increase the JEE stability, our perspective work relies on two axes both different and complementary. The first one is based on our modular approach, the second one consists in minimizing native code part of the execution environment by transferring it in Java language.

3.1 Exploiting modularity

Besides the ability to specialize some parts of the JEE, the modular approach can be exploited in several ways.

3.1.1 Reducing software errors

In order to reduce software errors the JEE can contain, it is possible to design integrity tests for each service (or a subset) of a module, for a subset of modules and for the whole JEE. This will increase the stability of the overall software. We already have a test suite for Scratchy and we are currently designing more service tests.

Thanks to the automatic generation of call stubs between services, it is also possible to introduce tests to dynamically check whether data pointers are correctly initialized.

At last, because SDE generates data type representation, we can assign to each type a magic word (see figure 4) and check dynamically whether every pointer corresponds to the correct kind of data type (for parameter passing or when accessing a structure). Of course a memory expansion and an execution time overhead are introduced, but this feature of SDE is only used to stabilize the JEE.

3.1.2 Transferring operating system code in modules

As SDE provides a good support to increase the reliability of the Java execution environment software, we could transfer some operating system code parts in modules. Of course, some extensions to SDE need to be done because the current version of SDE is specific to JEE generation. At present, we are working on a small operating system designed to execute Scratchy. This small OS provides basic services for memory, file system and device management.

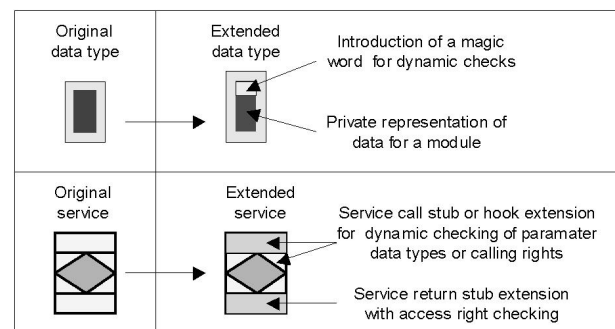


Figure 4. Extending SDE generation.

3.1.3 Increasing security

Several degrees of security, rights or confinement policies can be introduced at the module level and/or at the service level (see figure 4). Each module and/or service could specify the security according to its need. Security policies can be realized in several ways:

- dynamic check of rights before calling a service or using a data type structure belonging to the module. For example, a security policy can be introduced in order to dynamically check rights of native method invocation,
- use of an encryption protocol using keys to communicate parameters between services,
- use of a private representation of data in a module. For example all data of a module could be crypt and only

services belonging to this module can understand the representation.

3.2 Minimizing native code

The second perspective of our approach consists in transferring some native code of the JEE software in Java language aiming at exploiting the software confinement features of Java to increase the overall stability of the platform. At the opposite of current Java operating system projects [2] like J-Kernel, we do not want to enable the direct use of pointers in Java. So, our approach is to provide a java view of low-level resources without any pointer abstraction.

To validate that point, we have exported the native scheduler of Scratchy in Java. Hence, we designed an abstraction for interruption management and threads. The interface *OsInterrupt* specifies an interruption handler and provides a method called handler which is invoked when an interruption occurs. An implementation of this interface can be given at any interruption level. Each implementation has to provide its running parameters through Java fields. For example, a keyboard interruption implementation must define a field representing the value of the pressed key. This value is set using a minimum native interruption handler. In this way, the access of low-level IO control ports of the device is not needed and we still guaranty that no pointer is needed to write any interruption handler in Java side. The class *OsTimerInterrupt* is designed to be associated to the low level timer interrupt and enable the scheduler to realize its policy.

The *OsThread* class abstracts a Java thread context and provides the scheduler a *setThreadActive* method to execute its elected *OsThread* on the CPU. The scheduler performs a round-robin policy that manages all Java threads. It was very easy to be implemented and we showed that it is possible to transfer native code of the JEE in Java world. Our ongoing work focuses on transferring more complex JEE native software (including drivers) like graphics native APIs, sound APIs and also the garbage collector, in order to obtain the smallest native operating system needed for a JEE.

4 Conclusions

Our direction to build a dependable Java execution environment relies on two approaches. First, we exploit and extend the modular features of our JEE. Second, we minimize pieces of native code. This direction is valid only if complementary works aiming at eliminating Java software faults by using other languages or compilation techniques are made and if an effort to low-level design techniques like [6] are done.

At last, transferring operating system features in Java world is conceivable only if the JEE provides good per-

formance. Recent works made in this area [7, 1, 4, 9] are promising on that point and it is clear that Java has a place in embedded environments in the future.

References

- [1] Ajile. Overview of ajile java processors. Technical report, Ajile (<http://www.ajile.com>), 2000.
- [2] G. Black, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau. Techniques for the design of java operating systems. In *proc. of the USENIX Annual Technical Conference*, June 2000.
- [3] G. Cabillic, J. Lesot, M. Banâtre, F. Parain, T. Higuera, and V. Issarny. *The Application of Programmable DSPs in mobile Communications*, chapter A Flexible Java Environment for Wireless PDA Architectures based on DSP Technology, pages 119–135. WILEY press, december 2001.
- [4] Imsys. Overview of cjpeg processor. Technical report, Imsys (<http://www.imsys.se>), 2001.
- [5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *proc. of the 15th European Conference of Object Oriented Programming (ECOOP)*, Budapest, Hungary, June 2001.
- [6] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An idl for hardware programming. In *proc. of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, San Diego, CA, October 2000.
- [7] Nazomi. Boosting the performance of java software. Technical report, Ajile (<http://www.ajile.com>), 2002.
- [8] A. Reid, M. Flatt, L. Soller, J. Lepreau, and E. Eide. Knit: Component composition for system software. In *proc. of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 347–360, San Diego, CA, October 2000.
- [9] T. Suganuma, T. Ogasawara, T. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Systems Journals*, 39(1), February 2000. Java Performance Issue.
- [10] Sun Microsystems. *CLDC and the K Virtual Machine (KVM)*, 2000.
- [11] Texas Instruments. *TMS320C5x User's Guide*.

An Online Evolutionary Approach to Developing Internet Services

Mike Y. Chen, Emre Kıcıman, and Eric Brewer

mikechen@cs.berkeley.edu, emrek@cs.stanford.edu, brewer@cs.berkeley.edu

Abstract

High dependability in Internet services is a difficult challenge: new features are constantly added, the systems are being scaled to support more users, and these systems are subject to unpredictable workloads and inputs. To deal with these challenges, operators must constantly adapt and evolve their system in response to its dynamic behavior. We argue that this online evolution is necessary for the development and deployment of dependable Internet services. This paper presents a conceptual model of online evolution consisting of three phases—monitoring, analysis, and modification—and present techniques we have found useful in speeding the process of online evolution.

1 Introduction

There are many challenges in providing highly dependable Internet services. The fast software release cycles and growth rate of Internet services violate one of the traditional dependable computing principles of minimizing change. In practice, the fast release cycles mean that the software is far from perfect. In addition, the frequent software and hardware updates to the systems increase the probability of operator errors. Worse still, Internet services are exposed to unpredictable workloads [1, 7]. The fragility of existing Internet services is evident in many publicized outages [20, 21], and there may be many more unreported cases where only capacity or correctness was affected.

Traditional design and testing paradigms which focus on the pre-deployment phases of a service are not sufficient to cope with runtime problems caused by immature software, unexpected workload, and operator errors. In [13], Gribble argues that robustness based on precognition of the failure modes of a system is bound to fail. Even the most careful design will not correctly predict and handle all incarnations of hardware, software, operator and environmental faults.

To deal with this in practice, service operators and developers are continually monitoring their system's dynamic behavior, and adjusting its configuration and code to compensate for unexpected occurrences. In effect, Internet services are constantly being evolved in response to their operating behavior, albeit in an often ad hoc manner.

We argue that online evolution, *the constant adaptation*

and development of online systems, is a necessary part of the development and deployment of a service—as necessary as debugging and testing of code. Because we cannot simulate the unknown environment that a system is subject to, we have to use the real world as a continual testing environment, and be prepared to change the system online as we encounter unexpected occurrences and changing requirements.

Conceptually, online evolution consists of three phases, suggested by classic control theory:

1. *Monitoring* collects data on the behavior of a live system. Logging this information provides an analyzable history of the system's behavior.
2. *Analysis* distills the collected data to provide insights about the system, such as helping detect and diagnose faults.
3. *Modification* of a system is made to recover from a fault or adapt to meet new system requirements. It can be as simple as restarting a failed machine, or as complicated as rolling out a new version of the system software.

Online evolution includes short-term recovery from faults and adaptation to workload. Here, the evolutionary cycle of monitoring, analysing and modification can often be fully automated. Online evolution also extends to long-term development of features and architectural design. In these cases, evolution can only be partially automated with the system providing as much aid to the developer as possible.

In the second section of this paper we present a thorough discussion of the online evolutionary model of system development. The third section provides detail into the monitoring, analysis, and change techniques we have found useful in speeding the process of online evolution, and gives a brief overview of prototypes we have developed. The final sections discuss related work and summarize.

2 Online Evolutionary Model

Online evolution encompasses both adaptation to short-term fluctuations in the operating environment, and evolution to meet long-term trends. Operators and developers of Internet services evolve the system in response to their believed perception of the service's efficacy. In effect, they are creating manual feedback loops, using their knowledge of the system's operation to guide its short-term adaptation and long-term evolution. Today, this online evolution is at best ad hoc;

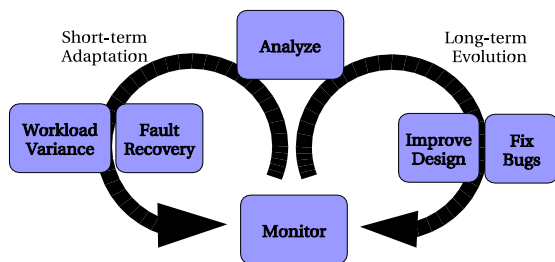


Figure 1. Short-term and long-term feedback loops

there is little common methodology for analysis and modification of the system—the monitoring, analysis and adaptation frameworks that do exist tend to be service-specific [22].

To systematize this approach, we use the notion of a “feedback loop” to express the relationship between a kind of problem, monitoring and analysis techniques that detect the problem, and adaptation or evolution procedures to fix it. For example, large spikes in workload can be detected via request counter. A service can then be adapted to handle this spike by adding more computing resources to the service. These feedback loops operate at various timescales: short-term adaptations may take seconds to minutes, and longer-term evolutions operate on the order of hours, days or more. As shown in Figure 1, here are four of the feedback loops that improve the availability and dependability of a service.

Workload Variance: Adapting to workload variance helps the service avoid overload or minimize resource consumption. Typically the performance of the system is analyzed and standby capacity is added to the service under saturation. Also, graceful degradation techniques such as harvest/yield trade-offs [11] and admission control can help the service maintain acceptable service quality.

Faults: Because hardware faults and software bugs will always occur, it is important that a system be ready to quickly mask and recover from faults as they occur. One of the biggest challenges is fault detection and diagnosis (i.e., to pinpoint what and where the faults are in the system) in order to recover. Simple faults, such as hard disk failures and fail-stop frontends are easily detected and recovered automatically through built-in redundancy, such as RAID. Other faults require more sophisticated monitoring, analysis and recovery techniques, and often require human intervention.

Software Bugs: In addition to short-term recovery from faults, it is often desirable to take more time to fix the source of the failure. Staged rollout [22] enables developer to use server logs, bug reports, and real user feedback to help detect and fix bugs before the software is

deployed across the whole service. A more observable system complements this practice because it helps the developers understand the internals of a system and reduces debugging time.

Design: On a longer time-scale, developers are interested in improving the architectural design of the system. Through online monitoring and analysis, developers can verify their own assumptions against the behavior of the deployed system, and improve on problematic designs.

In addition to these feedback loops for making systematic improvements to a service, there are feedback loops guiding the development of the service content, user interface, and feature list. This kind of development uses another kind of systematic feedback, analysis of user behavior and user feedback. Though of obvious importance, these other feedback loops are outside the scope of this paper.

3 Towards Rapid Evolution

In this section, we discuss the systematic introspection and analysis techniques that we have found useful in aiding rapid online evolution. In addition, we discuss how some feedback loops can be fully automated using known adaptation techniques. Finally, we present some initial results from in-development prototypes of these techniques.

3.1 Monitoring

Building measurement infrastructure into a system is the first step in enabling developers and operators to monitor and diagnose a live system [4]. As argued in [9], the introspection framework should be built in an application-independent fashion to simplify application development, eliminating the need for programmers to insert explicit logging calls.

Though most systems have some concept of logging their actions and state, most do not log enough information to fully understand a system’s dynamic behavior. For example, Apache logs externally visible events such as URL, IP address, and timestamp but does not record the internal components used. It relies on programmer-inserted error messages to aid fault diagnosis.

The following are three classes of information that we believe are useful to record to help expose a system’s behavior. Most monitoring systems today only focus on performance, ignoring other useful information.

Inter-component and inter-request relationships: By tracing a service request through a system and logging all components used, we can discover the dynamic functional dependencies between the components. Also, in Internet services where end-user interactions span multiple HTTP requests, we must track state dependencies between these HTTP requests to understand how the system behaves end-to-end in a session. By logging reads

and writes to state, we can discover many of these inter-request dependencies. This information can be used to catch unintended interactions among components, as well as to aid fault diagnosis, debugging and system redesign.

Inputs to the system: Logging the inputs to a system is useful for discovering failures due to pathological inputs, and for use in future regression tests. Logging can be done efficiently as a circular buffer to keep only those inputs that co-occur with failures. Another use of this information is for anomaly detection. For example, different browsing behavior from users of a particular version of web browser might indicate HTML compatibility problems.

Performance characteristics: Logging performance information, such as throughput and response time, is useful for detecting resource exhaustion, configuration errors, or workload spikes.

3.2 Analysis

The goal of analysis is to distill large amounts of data into useful information that help automate adaptation or help developers and operators understand the systems to evolve them correctly.

Anomaly detection: Statistical and machine learning techniques can be used to detect unusual situations, such as performance degradations, unexpected interactions or behaviors, that most likely indicate faults or bugs [10, 14, 19, 15]. Today, these systems are used to detect anomalies in the behavior of a single node across time. In a replicated Internet service, we can extend these same techniques to compare replicated peers in the system to detect anomalies.

Dynamic visualization: dynamic dependency graphs help developers and operators understand the real behavior of a system during testing and deployment. Functional dependency graph represents inter-component relationships and state dependency graph represents inter-request relationship.

Fault diagnosis: once we have detected a believed failure, either using anomaly detection or through direct observation, we can use statistical analysis techniques such as data clustering to correlate the failures with logged events in the system. For example, we can correlate failures with physical components or interactions between components to quickly identify the root-cause of failures automatically. [9]

3.3 Modification

The ability to modify systems online without bringing down the services is critical for services that requires high

availability. In practice, modification only happens online and automatically for simple workload variance and simple faults, such as failed disks. Most failures and performance degradation still involve operators and developers in the feedback loop.

After analyzing and forming a hypothesis of the system's behavior, we can close some of the feedback loops by providing a trigger for dynamic adaptation techniques. For software bugs, recursive restarts [6] bring the system back to a known, functioning state. For configuration errors, undo [5] helps the system configuration rollback to a previous working configuration. For an overloaded system, dynamic connection management [8] allows it to degrade gracefully by performing admission control or by making harvest/yield trade-offs.

3.4 Initial Results

As an initial step towards an infrastructure that supports rapid online evolution, we have built two tools, Pinpoint and Connection Manager, to improve the monitoring and modification stages of the fault recovery feedback loop.

Pinpoint is a tool that uses aggressive logging and data clustering analysis to automate fault diagnosis in Internet services [9], thus improving the monitoring stage of the fault recovery loop by speeding the time to detect and diagnose faults.

The Pinpoint prototype is implemented on the Java 2 Enterprise Edition (J2EE) platform for Internet services, and requires no modifications to be made to a J2EE application. It dynamically traces real client requests through the system. We were able to automatically identify the root causes of single-component failures 80-90% of the time with an average rate of 40-50% false positives without any application-level knowledge of the components and the requests. This rate of false positives is significantly better than other common approaches that achieve similar accuracies.

Connection Manager (CM) [8] is a management layer on top of load-balancing switches that enables applications and the infrastructure to control how external connections are mapped to internal resources. CM improves the modification stage of the fault recovery loop by quickly unmapping failed resources, and allowing online reinsertion of repaired resources.

We have used our CM prototype to implement several service-independent adaptation techniques. We are able to automatically perform dynamic resource allocation and rolling reboots for several real Internet services, including web, email, and instant messaging, with no dropped connections. In addition to unmapping failed resources in less than 2 seconds, CM also helps services degrade gracefully under overload by using admission control to limit incoming connections.

We are currently working on merging the two tools together to further automate the fault recovery loop, as well as investigating how the improvements Pinpoint and Connection Manager make to the monitoring and modification stages of the fault recovery loop might also be applied to improve other

feedback loops as well.

4 Related Work

Online evolution in Internet services has important similarities to the spiral model of software development [3]. Both emphasize the feedback loop from system behavior and requirements to development. There are two important differences between the two, however. First, there is an order of magnitude difference in the rate of change under the two models. Change in an Internet service has to happen constantly: adaptation to changing workloads and faults must often occur within minutes; bug-fixes are propagated daily; and new features are added almost as often. The second major differentiator is that online evolution provides for the existence of multiple concurrent feedback loops.

Though several projects have focused on engineering of web services, most consider service development as being separate from deployment and maintenance [17], and only a few acknowledge the need for constant, online change [16]. In [12] Gaedke presents a model of *Web Engineering*, evolution of web applications using component-based software, but focuses on component reuse and enabling the web service code to be changed and extended over time. We are not aware of any work that recognizes the existence of multiple feedback cycles in Internet service development and deployment.

Several projects share our view in helping systems adapt and evolve online, with each focusing on a subset of the feedback loops we have identified in section 2. OceanStore [18] and Hippodrome [2] are storage systems with automated adaptation. Focusing on the workload and fault feedback loops, OceanStore adapts to changes in the system, such as server addition and removal, to minimize management overhead and maintain data persistence. Hippodrome closes the workload feedback loop by automating the design and configuration process of a storage system to iteratively reconfigure itself in response to workload requirements. [23] automates the workload feedback loop and applies admission control to improve the performance of Lotus Notes under saturation.

5 Summary

Online evolution is a necessary part of making Internet services robust to unexpected occurrences and changes in system requirements. In this paper, we presented a conceptual model for online evolution based on monitoring, analysis, and modification; and discussed techniques for improving the online evolutionary process through aggressive logging, and systematic analysis and modification techniques.

We have built our initial prototypes and evaluated these strategies. Our early results are promising. We are currently working on extending our prototypes to encompass our complete model of online evolution; and providing support for the online evolution process as part of the management infrastructure for Internet services.

References

- [1] S. Adler. The Slashdot Effect: An Analysis of Three Internet Publications. *Linux Gazette*, 38, March 1999.
- [2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, , and A. Veitch. Hippodrome: running circles around storage administration. In *Conference on File and Storage Technology*. USENIX, 2002.
- [3] B. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(2):61–72, May 1988.
- [4] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiawicz, and D. Patterson. ISTORE: Introspective Storage for Data-Intensive Network Services. In *HotOS-VII*, 1999.
- [5] A. B. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to Dependability. In *10th ACM SIGOPS European Workshop*, 2002.
- [6] G. Candea and A. Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. In *HotOS-VIII*, 2001.
- [7] Computer Emergency Response Team (CERT). CERT Advisory CA-2000-01: Denial-of-service developments, 2000. <http://www.cert.org/advisories/CA-2000-01.html>.
- [8] M. Chen and E. Brewer. Active Connection Management in Internet Services. In *Eighth IFIP/IEEE Network Operations and Management Symposium*, 2002.
- [9] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Symposium on Dependable Networks and Systems (IPDS Track)*, 2002.
- [10] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *18th ACM Symposium on Operating Systems Principles*, 2001.
- [11] A. Fox and E. Brewer. Harvest Yield and Scalable Tolerant Systems. In *HotOS-VII*, 1999.
- [12] M. Gaedke and G. Graef. Development and Evolution of Web Applications using the WebComposition Process Model. In *International Workshop on Web Engineering at the 9th International World-Wide Web Conference*, Amsterdam, the Netherlands, May 2000.
- [13] S. Gribble. Robustness in Complex Systems. In *HotOS-VIII*, 2001.
- [14] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *International Conference on Software Engineering*, June 2002.
- [15] Joseph L. Hellerstein. A General-Purpose Algorithm for Quantitative Diagnosis of Performance Problems. *Journal of Network and Systems Management*, 2001.
- [16] D. B. Ingham, S. J. Caughey, and M. C. Little. Supporting Highly Manageable Web Services. In *WWW9*, April 1997.
- [17] E. Kirda, M. Jazayeri, and C. Kerer. Experiences in Engineering Flexible Web Services. *IEEE Multimedia*, 8(1):58–65, January 2001.
- [18] J. Kubiawicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gum-madi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of ACM ASPLOS*. ACM, 2000.
- [19] W. Lee and S. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, 1998.
- [20] News.com. E-tail sites crash over holiday weekend, 2001. <http://news.com.com/2100-1017-249048.html>.
- [21] News.com. Ebay stumbles with outage, 2002. <http://news.com.com/2100-1017-860703.html>.
- [22] D. Oppenheimer and D. A. Patterson. Architecture operation and dependability of large-scale Internet services. In *Submission to IEEE Internet Computing*, 2002.
- [23] S. Parekh, N. Gandhi, J. L. Hellerstein, D. Tilbury, T. S. Jayram, and J. Bigus. Using Control Theory to Achieve Service Level Objectives in Performance Management. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

Applying source-code verification to a microkernel — The VFiasco project

— Extended Abstract —

Michael Hohmuth
Hendrik Tews

Dresden University of Technology
Department of Computer Science

vfiasco@os.inf.tu-dresden.de

Shane G. Stephens

University of New South Wales
School of Computer Science and Engineering

Abstract

We present the VFiasco project, in which we apply source-code verification to a complete operating-system kernel written in C++. The aim of the VFiasco project is to establish security-relevant properties of the Fiasco microkernel.

Source-code verification works by reasoning about the semantics of the full source code of a program. Traditionally it is limited to small programs written in an academic programming language. The project's main challenges are to enable high-level reasoning about typed data starting from only low-level knowledge about the hardware, and to develop a clean semantics for the subset of C++ used by the kernel. In this extended abstract we present our ideas for tackling these challenges. We focus on a type-safe object store that is based on a hardware model that closely resembles the IA32 virtual-memory architecture and on guarantees provided by the kernel itself. We also briefly touch on the semantics for C++.

Please find the full version of this paper at <http://www.vfiasco.org/objstore.pdf>.

1 Introduction

The VFiasco project aims at the mechanical verification of security-relevant properties of the L4-compatible Fiasco microkernel [2].

The goal of the project is an operating-system kernel that provides *verified* security guarantees. Such a kernel could be used as a basis for applications with high-level security requirements. Verification is a very expensive process (both in man power and time); for success it is crucial to minimize the size of the system. Huge bug-afflicted monolithic kernels are outside the scope of current verification technology. On the other hand, microkernels are the smallest kernels that provide an anchor for building secure systems: separate protected address spaces. Therefore, they are the best choice for constructing a verified secure system.

VFiasco is a work-in-progress. In this paper we report

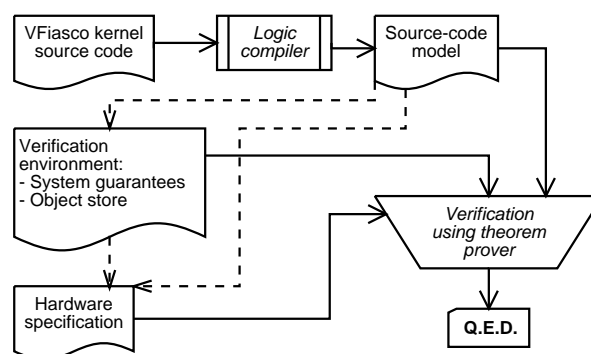


Figure 1. The verification process. Legend: Solid arrows show the flow of data. Dashed arrows indicate a «uses» relationship.

on one aspect of the project: the modeling of a type-safe object store on top of a model of virtual-memory hardware.

To our knowledge, the VFiasco project is unique in scope and intended thoroughness. We aim at modeling all of the kernel's source code in very fine grain, and we intend to “run” this software model on a hardware model that closely resembles real hardware. These qualities are meant to establish an as-yet unseen level of confidence in our software. Our formal-verification approach exceeds even what is necessary to fulfill the development requirements of the Common Criteria's¹ highest assurance level, EAL7.

Fiasco has been implemented in C++. For the verification we develop a dialect of C++ with a precise semantics, which we call “Safe C++.” The verification will be carried out in the interactive theorem prover Isabelle/HOL [8]. This theorem prover uses higher-order logic (HOL) as its input language. Therefore, we translate the kernel's source code from Safe C++ into its semantics expressed in HOL. In our

¹The “Common Criteria for information Technology Security Evaluation” (CC; ISO 15408) replaces the Trusted Computer System Evaluation Criteria (TCSEC; better known as the “Orange Book”) in the U.S.A. and Information Technology Security Evaluation Criteria (ITSEC) in the E.U.

approach, a *logic compiler* performs this translation automatically. This technique is in stark contrast to approaches in which parts of the source code are translated manually to a more or less abstract model. Figure 1 illustrates our verification methodology.

The basis of the semantics of Safe C++ is a model of the computer system, which we must provide in the theorem prover. An important problem in the project is to find the right abstraction level for this model. To facilitate the verification, we would like to have the abstraction level of a virtual machine that provides a type-safe object store—a memory that supports reading and writing of typed values and that guarantees safe accessibility of these values (as well as other properties). Such an interface would allow us to reason on a comfortably high level, ignoring the complexity of contemporary virtual-memory systems and memory allocation.

However, we cannot simply assume such an object store before verifying the Fiasco microkernel. Fiasco executes in a much more hostile environment—on virtual-memory hardware. In fact, one of the kernel’s tasks is the provision of guarantees that allow the construction of such an object store in the first place. Therefore, the existence of an object-store layer with strong properties should be a proof goal, not a base assumption.

In this paper, we fill the gap between high-level programming languages (in our case Safe C++), which provide safety by means of protecting typed memory objects from arbitrary accesses, and contemporary hardware with virtual memory. We develop a type-safe object store based on a set of memory models that mimic the way a high-level-language programmer thinks of memory, but still can be implemented using a concrete CPU model. Using these memory models, it is possible to reason about the Fiasco kernel, ignoring the current virtual-memory setup and the effects of page faults on the program state.

2 Related work

Model checking. Model checking has been successfully applied to several systems in the past [9]. However, this technique can only be applied to abstract models of real systems, because the state space of the real systems is too large. This restriction limits the conclusions that can be drawn from model checking. For instance, in [9] Tullmann and colleagues verify liveness properties of the Fluke kernel’s IPC subsystem. Thereby they abstract away from the actual data that is transmitted. While they actually proved the absence of deadlocks, it is theoretically possible that the IPC subsystem deadlocks because it dereferences a user pointer (which has been abstracted away in the model checker).

Proof-carrying code. Proof-carrying code [7] solves the problem of executing untrusted (user-supplied) code in kernel mode. In this approach, the kernel accepts only those

extensions that are accompanied with a valid proof for a given security policy. In a typical application (for example, a network filter) the involved verification is trivial and can be automated.

In the VFiasco project, we tackle a rather different problem: proving the kernel *itself* correct. The problem of safely extending this kernel is orthogonal to our work.

Static source-code checking. There are many tools in the spirit of `lint` that statically analyze source code. For instance the tool presented in [1] has been used to find many bugs in the Linux kernel. Static source-code checking is different from testing in that it analyzes the source code instead of running the system. With testing it has in common that it assists in finding programming errors. In the VFiasco project our concern is not so much to find errors, but to *give guarantees* about their absence.

Theorem proving. There are a few projects that apply theorem proving at the source code level as we do.

In [6] Liu and colleagues use the theorem prover Nuprl to verify the correctness of network-protocol stacks and to optimize such stacks. There are two major differences to the VFiasco project. First, to enable the verification the original C source code was rewritten in a carefully chosen subset of the functional language Ocaml [5]. In contrast, we plan to develop a semantics of a subset of C++ that essentially contains everything needed for kernel programming, including abrupt termination², `longjmp`’s, and pointer arithmetic. Second, Liu and colleagues do *not* verify the source code. Instead, they verify program transformations.

Our approach to a semantics of C++ is very similar to the one used in the LOOP project for Java [4]. We also use coalgebras to represent statements and expressions. The LOOP project focuses on the verification of Java applications, therefore they can use an object memory that directly represents Java objects [10]. A central aim of the VFiasco project is to incorporate system internals like page fault handling and protection levels into the verification. Therefore we need a more low-level view on the object memory.

3 Verification approach

This section sketches the main ideas of our semantics of Safe C++. Please see the full version of this paper for more details, including our approach for dealing with Fiasco’s inline-assembler statements [3]. The semantics of Safe C++ is based on two main ideas: *state transformers* and *under-specified functions*.

State transformers. With State transformers (also called *coalgebras*) we adapt the approach of [4] to C++. State transformers allow us to give a relatively simple semantics

²An expression or statement terminates *abruptly* if the control flow does not reach the end of the statement or expression because, for instance, a `break` or `return` was executed.

to statements like `break`, `continue`, and even `goto` and the library routines `setjmp/longjmp`. A state transformer is a function (in the mathematical sense) of the following type:

$$\text{St} \longrightarrow \text{ExprResult}(\text{St})$$

Here St stands for the set of all possible states of the system (we elaborate more on structure of St in Section 4). The type $\text{ExprResult}(\text{St})$ is a disjoint (or tagged) union that models the different possible results of C++ expressions (or statements). For instance, if an expression does not terminate, then its result is the distinguished element $\text{Bug} \in \text{ExprResult}(\text{St})$; a `break` statement in a state $s \in \text{St}$ yields $\text{Break}(s)$.

All C++ expressions and statements (including complex statements) are modeled as state transformers. Composition of state transformers is defined such that the second state transformer is skipped if the first one does not terminate normally. Special statements that regulate flow control are modeled with functions that manipulate state transformers and their results. For instance, loops are wrapped into a function that translate a result of $\text{Break}(s)$ into a normal state, thus resuming execution with the statement that follows the loop.

Underspecified functions. An underspecified function is a function that, although some properties are known, the precise result when applying them is not specified. Thus, in the theorem prover one can only work with the known properties and not with the result of the application. We use underspecified functions to generate the locations of variables and to transform typed values into their byte representation. Underspecified functions allow us to include pointer arithmetic and unsafe type casts in Safe C++.

4 Type safety and virtual memory

In this section, we discuss what a Safe-C++ program's state St contains and which operations it supports. This interface comprises the “architecture” for which our logic compiler produces “code.”

It is possible to apply the state-transformer approach we presented in Section 3 to environments with widely differing abstraction levels. For VFiasco our goal is to keep a high-level-language programmer's view during verification while still enabling reasoning about low-level hardware manipulation.

Programmers of high-level languages such as Safe C++, including kernel programmers, make many assumptions about the environment in which their program eventually runs. Table 1 lists a number of such assumptions, which we call *object-store properties*. For example, programmers assume that a program can successfully access typed objects that have been properly allocated.

Unfortunately, a storage model that is *a priori* type safe is not adequate for modeling a kernel environment for two

reasons. First, such an assumption might be wrong—invalidating all verification results—because there is no system component that provides type safety. In the real world, the kernel runs on top of an untyped virtual memory and must ensure its own type safety. Second, kernel programmers sometimes need to circumvent the compiler's type safety for low-level systems programming, for example for manipulating CPU data structures.

Therefore, instead of assuming object-store properties from the start, our approach is to prove them starting from low-level knowledge. In summary, we aim for the following design goals in modeling our object store:

Credibility. We want to start only from very basic low-level assumptions. Therefore, the storage model should be based on a memory model that closely resembles the virtual-memory hardware on which the kernel executes. Further, we must document all base assumptions³ we make about the hardware and the Safe-C++ compiler. We describe our hardware model in Section 4.1.

Type-safe object store. Efficient interactive reasoning about a program requires high-level knowledge of the program's state. Therefore, we need to create a verification environment that provides a type-safe object store with proven object-store properties. This environment consists of a mapping of an object-store interface to a virtual-memory interface. Section 4.2 describes our verification environment.

Direct hardware access. It must be possible to circumvent the object store and access virtual memory directly. We address this requirement in the full version of this paper [3].

There are also a number of second-level design goals:

Reusability. The object-store specification needs to be generic enough to serve as the general target language of the logic compiler. Fiasco's high-level *and* low-level kernel code as well as boot code should be expressible. In the future, we also would like to use it as a target for user-program code. Section 4.2.1 explains how we achieve this goal.

Automation. Based on the object-store properties, we need to provide powerful theorem-rewriting rules that automatically simplify logic-compiled source code without operator intervention as far as possible. We briefly discuss our rewriting rules in Section 4.2.2.

4.1 Hardware model

The hardware model provides the basis for the semantics of Safe C++. It defines the set of system states St and primitive operations, like reading in memory and inserting page mappings. A complete model of the Intel IA32 architecture is far beyond our project. Rather, we use an abstraction of the hardware that contains just those primitive operations that are necessary to run the Fiasco microkernel.

Currently, the model consists of four main components: the *Physical memory*, the *TLB specification*, *Page-fault han-*

³In our verification, these base assumptions play the role of axioms.

Assumption (object-store properties)	Reality (low-level knowledge)	Implied system guarantee
All program code and properly allocated data are accessible	Any memory access can fault during a TLB or page-table access	Pinned memory, or kernel faults in “correct” memory; kernel is mapped into all address spaces
Objects do not change value unless updated explicitly	different objects might overlap; the same object might be mapped twice	All objects are allocated such that no two object’s virtual-address regions overlap
Program reads and writes typed objects	Objects are stored in byte sequences; the byte representation of most data types is unknown to the programmer	There exist two inverse functions that convert between typed values and byte sequences
Program operates in flat virtual address space	Program code and data are split into pages, some of which are stored noncontiguously in physical memory, and some of which are not memory-resident	Page-fault code and virtual address space maintain “illusion” of flat address space

Table 1. Examples of high-level-language programmer’s assumptions and guarantees needed from the memory subsystem. Usually, programmers assume object-store properties like those in the left column. However, these properties are not true in general. In reality, facts like those in the middle column can falsify the assumptions. The object-store properties are valid only if the runtime system provides the guarantees in the right column.

ding, and functions for *reading from and writing to virtual memory*. As the VFiasco project progresses we expect the hardware model to become more detailed, for instance by modeling interrupts and protection levels.

4.2 Verification environment

In this section, we construct a type-safe object store, assuming only a model of virtual-memory hardware.

4.2.1 Encapsulating system guarantees

System specifications. We have been able to prove the object-store properties using system properties like those in Table 1’s “implied system guarantee” column.

As a means for structuring the proofs, we have factored the system guarantees into a number of *system specifications*. The extent of these guarantees differs between low-level and high-level parts of the kernel. For example, the kernel’s page-fault handler can access only some parts of the kernel’s virtual address space, and it is not allowed to page-fault recursively. We therefore have taken care to allow the specifications to be parameterized with memory regions that can be safely accessed. Here we discuss two of these specifications: *Plain Memory* and *Allocator*.

The Plain Memory specification models a flat virtual address space in which bytes can be read or written. This specification provides the notion of *blessing* memory regions. It asserts that reading or writing to a memory region that is read-blessed or write-blessed respectively does not fail. The object-store properties are valid generally only for objects residing in blessed memory. We call instances of this specification a *memory model*.

Normally, these memory models must be implemented

in terms of the hardware model’s virtual-memory interface.⁴ Therefore, each memory model uses one particular page-fault handler.

The Allocator specification contains operations for allocating memory blocks in blessed memory. It asserts that within blessed memory regions, each allocated block is accessible at only one virtual address. This property facilitates safe object reads and writes. There are a number of instances of Allocator provided by Safe C++—in particular the static allocator and the stack allocator; for a kernel, there is no predefined heap allocator. However, there can be any number of user-defined allocators written in Safe C++.

Instantiating the system specifications. For each part of the kernel that is to be verified, we must instance all system specifications that are to be used: one memory model and potentially multiple Allocator instances. For the lowest-level parts of the kernel, these instances only include axiomatic knowledge about builtin Safe-C++ allocators and about the memory state after boot-up. Higher-level parts can use a richer set of Allocator instances and a more complex memory model that uses a Safe-C++ page-fault handler verified as a lower-level part.

Our memory models are of particular interest because they allow us to use the object-store interface for both low-level and high-level kernel code. In the remainder of this section, we discuss the two memory models we use for these two types of kernel code. We have proven that these memory models are indeed instances of Plain Memory.

The “Simple VM” memory model. This memory model is used for verifying low-level kernel code. Its read and write operations are based on our hardware model. In

⁴However, there are other memory models that are conceivable as well: For example, during the boot process, paging may be turned off, which results in a memory model that operates directly on top of physical memory.

this model, each invocation of the page-fault handler is considered an error. Blessings are based on the contents of the current page table.

Based on the invariant that the kernel's code and static data are always mapped⁵ and on the precondition that there is an accessible stack, the Simple VM model can run code that does not rely on page-fault handling and that does not need a custom allocator. We use this model to verify Fiasco's page-table insertion, low-level allocator, and page-fault handler functions.

The “Kernel Memory” memory model. For the bulk of Fiasco kernel code, the Simple VM model does not contain enough features. In particular, it lacks dynamic memory allocation, kernel-virtual memory manipulation, and lazy page-directory updates. Fiasco relies on these features when it dynamically allocates data structures such as thread descriptors from its private memory pool. In this event, it maps new pages into a “master” virtual-address space and lazily updates the kernel regions of user tasks' virtual address spaces from the master copy upon page faults. These lazy updates are completely transparent to the kernel code; for this code, it looks as if the allocated memory “is always there.” We reflect this view in our memory model “Kernel Memory.”

In this memory model, read and write operations again are based on our hardware model (Section 4.1). The behavior of these operations is similar to the Simple VM model; however, here page-faults invoke the global page-fault handler.

In addition to the Simple VM blessings, the Kernel Memory model also regards as blessed the memory blocks that were allocated using the low-level allocator. Based on this low-level allocator, we can verify a hierarchy of more complex allocators (such as Fiasco's slab allocator).

4.2.2 The object-store layer

The object-store layer is the interface that provides the desired object-store properties. It provides functions for safely manipulating typed objects. This interface is the target language used by our logic compiler.

This layer relies on the guarantees provided by previous section's system specifications. As the object-store layer is independent from the concrete instantiation of these specifications, it works with both the Simple VM model and the Kernel Memory model.

Therefore, it is possible to logic-compile *all* kernel code towards the same object-store interface. Using this interface, we can verify even low-level Safe-C++ code such as the page-fault handler, which constitutes part of the Kernel Memory model. For this verification, we instantiate the Plain Memory specification using the Simple VM model,

⁵This invariant needs to be set up by the boot process.

which uses only hardware features and does not rely on other Safe-C++ code.

We were able to prove many object-store properties such as “Writing to some allocated object does not accidentally modify any other allocated object” and “After writing to an allocated object, reading from that object actually returns the value written.” These properties usually take the form of theorem-rewriting rules that allow semiautomatic simplification of and reasoning about state transformers that use only the object-store layer. When reasoning about a sequence of object-store operations, these rewrite rules help by removing uninteresting state modifications.

5 Conclusion

This extended abstract presents the main ideas for applying source-code verification to the Fiasco microkernel in the VFiasco project. The main challenge in this project is to enable high-level reasoning in terms of typed objects during the verification, yet assume only low level hardware properties. We solve this problem with several layers of parametrized specifications.

References

- [1] D. Engler, D. Yu Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, 2001.
- [2] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [3] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März 2002, Dresden University of Technology, 2002. Available from URL: <http://www.vfiasco.org/objstore.pdf>
- [4] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, 2000.
- [5] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system*, 2001. Available at URL <http://caml.inria.fr/ocaml/>.
- [6] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. P. Birman, and R. L. Constable. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating System Principles (SOSP)*, pages 80–92, Kiawah Island, SC, December 1999.
- [7] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 1997.
- [8] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer, Berlin, 1994.
- [9] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: A practical tool for OS implementors. In *Workshop on Hot Topics in Operating Systems*, pages 20–25, 1997.
- [10] J. van den Berg, M. Huisman, B. Jacobs., and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT '99*, number 1827 in LNCS, pages 1–21, 1999.

Back to the Future: dependable computing = dependable services

Jeffrey Chase, Amin Vahdat*, and John Wilkes

Abstract

Clients are coming to rely more and more on external services to meet the needs of their users, and the clients are increasingly simple caches of soft state – “truth” is maintained elsewhere. As a result, the user experience of dependability is better served by making those services ultra-dependable than by increasing the reliability of an individual client. We explore here some of the consequences of this statement, and conclude that developing scalable, dependable services may be a more fruitful approach than an extreme emphasis on “dependable OSes”. Along the way we look at quantifying “dependability” in this new world; some of what it takes to provide dependable, large-scale services; and some approaches that we are exploring to do so.

1 Back to the Future

Once upon a time, the US space program funded the development of ultra-dependable writing instruments. The result was an astounding piece of technology: a pen that could write upside down, in zero gravity, underwater, etc. It was priced accordingly. NASA’s extreme conditions required it. But the rest of us simply pick up a new, cheap pen when the one we are using stops working, and think nothing of discarding the old one. We find it more economical and convenient to rely on a service that supplies pens, not on a super-dependable pen.

The analogy above hints at our position on this topic: user perception of “dependability” is increasingly driven by the dependability of the underlying services rather than by the dependability of an individual client. Why is this?

Increasingly, clients are I/O devices rather than computing platforms. They are becoming more diverse, cheaper, and more specialized. And they are all connected to the service infrastructure. Any device that captures data (cameras, laptops, sensors, phones) has to preserve the data for later access by other services and applications. Any device that

presents data to a user (MP3 players, viewers, WAP phones, monitors) gets it from its environment.

The trend is driven by Moore’s Law and better connectivity: rather than simply becoming more powerful, client devices are trading some of that power for more portability. Further, they are increasingly communication-oriented devices rather than pure computing devices – which depends on connectivity, but connectivity is not the barrier to dependability that we once thought it was. In our homes and offices, broadband is as dependable as our phone and electrical systems, which we take for granted. On the road, we have good coverage from cellular phone networks and 802.11 in public places. And communication performance will improve dramatically as we pave over the “last mile.” At the same time, clients are acquiring increasing abilities to cache data, so temporary disconnections are less problematic.

Meanwhile, applications are increasingly server-based. There are several drivers for this, including:

1. Many services are intrinsically based on shared state (banking, commerce, trading, reservations, google, yahoo) or communication through resilient (albeit temporary) state (e.g., mail, messaging).
2. Growing numbers of services provide a place to upload data and share it with other people or with other devices owned by the same user.
3. Shared information is of intrinsically higher value than information that is usable only by one person; and that value is in proportion to the number of people who come together to take advantage of it.
4. We are becoming used to higher levels of dependability – we treat it as an unwarranted exception when an airline booking system doesn’t work, rather than the small miracle that it is when things go well.
5. The very portability of clients means that they are subjected to a greater range of threats than the fixed computing systems of yore: a device that can easily get dropped, stolen, or lost is not an ideal environment for preserving the only copy of valuable, long-term state.

*Contact Author: Amin Vahdat, Box 90129, Department of Computer Science, Durham NC 27708, vahdat@cs.duke.edu

We believe that the computing environment will evolve to one in which anything involving storage will be backed by reliable servers in controlled environments – and client devices will become ever more interchangeable, merely display devices and caches for data and software. This will bring its own challenges, such as consistency issues – clients will temporarily store data in write-back caches when they become disconnected, pending transmission to the service. Dealing with this will be an important service-architecture question – but all our experience with distributed file systems suggests that it will be manageable.

Consider the oft-used metaphor of the electricity supply, with a small twist. Client devices run on batteries, which are just a cache for electricity generated by the utility and delivered through the wall. Making the batteries more reliable and higher capacity does improve the quality of use, but ultimately we depend on that electric utility being there for us to do our work. But note that it's not a particular power source in that utility that we depend on: individual users typically do not care which site actually generated the electricity, they just want the power. Similarly, when users access a service, they care less and less about where it runs.

This is not a new idea: thin clients, network computers, and now scalable utility computing. But it is happening just the same, and participation from the operating system community is central to achieving the vision if we are to meet the levels of dependability that people are coming to take for granted across an ever-wider range of services.

2 Utility computing as the path to dependability

We believe that server-based computing and self-organizing resource utilities (server/network/storage farms) are the basis for dependable computing in the future. What will it take to realize this? At some level, much of it is simply good resource management, coupled with development of appropriate resource and service abstractions. A few things complicate this: the sheer scale (millions of clients, not tens or hundreds); the rapid rate of change of demand levels, enabled by the any-to-any connectivity offered by the Internet; the economics of supporting a utility-based infrastructure; and all the privacy, data integrity, security, and service-level predictability demands that “dependability” implies. We need to extend the Internet reliability and robustness model to services: we want to detect failures and route around them, as transparently to end users as possible.

Building robust services today requires cluster-based techniques where potentially thousands of individual machines deliver some higher level service (e.g., google's web search scheme). Similarly, geographic replication and transparent request redirection (e.g., using Akamai DNS servers) are employed to avoid network congestion and individual failures. All these techniques are transparent to end

users who simply request a service and are agnostic as to who actually delivers it. Such separation of service from a specific machine offers the promise of eliminating Lamport's Pitfall, where “A distributed system is one in which the failure of a machine I've never heard of can prevent me from getting my work done.”

This naturally leads to solutions that enable services to be dynamically provisioned – and then to dynamic resource provisioning for network, computational resources, storage, memory, etc. Furthermore, we look to schemes that allow business-driven levels of performance and dependability to be specified – and followed. To support this, we are increasingly able to provision sufficient resources “on demand”, quickly enough to deliver desired service levels under rapidly-changing loads.

This has been accomplished, in part, by better understanding of service-level agreements (SLAs). SLAs used in computer networks have demonstrated the benefits of using economic incentives to ensure well-provisioned services. That is, availability of sufficient resources is much more likely if delivering better performance and dependability results in more revenue.

Fundamental to our approach are the following techniques:

- The use of SLAs, both to quantify the desired goals, and to provide economic incentives for the utility providers.

We hope that providing cost models for resources will motivate application developers to deploy efficient software for a given demand level. Even if this is not the case, similar models can be provided at the resource-management layer.

- Mechanisms to allow the resource utility to provision to deliver target levels of performance and reliability. This includes mechanisms to prioritize resource allocations during temporary overload.
- Simultaneously performing replica placement, resource routing, and overlay topology configuration to achieve target levels of “performance” for minimal “cost”.
- Scalable algorithms for maintaining the utility through the aggressive use of caching, approximate information, hierarchy, and aggregation.
- Achieving robustness by deploying additional resources and redundancy. There are many examples of this principle, and they make server-computing inherently more dependable: RAID, dynamic replication, redundant paths, multipath routing, session recovery, edge caching and stashing, dynamic service placement and migration.

- Making all these techniques self-managing, so that people do not have to be involved in the systems' response to events (load changes, failures, etc.).

The above list applies mostly at the resource layer. It is also fruitful to consider application-level adaptations: ideally, they should be structured to be fluid, i.e., independent of the number and placement of servers and how load is divided among them. Applications should allow the system infrastructure (utility) to determine service placement, replication degree, and binding to peer services (databases, file servers) in a multi-tier structure. In this way, the utility can monitor conditions, adapt to failure, dynamically adjust placement and redundancy degree, scale up or scale back, and (re)allocate available resources to provide the best global service (for application-specific definitions of "best").

Between these two levels are frameworks that provide for application deployment, and adaptation to resource or application failures that can be accommodated by reassignment of resources to a service, and rebooting [6].

3 Examples of service utilities

3.1 Opus

Opus [2] is an overlay peer utility service. It allows individual applications to specify their performance and availability requirements. Based on this information, Opus initially maps applications to individual nodes across the wide area. Once this has been done, observed access patterns to individual applications are used to dynamically reallocate resources to match application requirements. For example, if many accesses are observed for an application in a given network region, Opus may reallocate additional resources close to that location.

One key challenge to achieving this model is determining the relative utility of a given candidate configuration. That is, for each available unit of resource, we must be able to *predict* how much any given application would benefit from that resource. Existing work in resource allocation in clusters [3] and replica placement for availability [10] indicate that this can be done efficiently in a variety of cases.

One key aspect of our work is the use of Service Level Agreements (SLAs) to specify the amount each application is willing to "pay" for a given level of performance. Opus uses utility functions for this: it makes allocation and deallocation decisions based on the expected relative benefit of a set of target configurations, based on an estimate of the marginal utility of resources across a set of applications at current levels of global demand [3].

Opus employs a global *service overlay* to maintain soft state about the current mapping of utility nodes to hosted applications (group membership). This service overlay is

key to many individual system components, such as routing requests from individual clients to appropriate replicas, and performing resource allocation among competing applications. Individual services running on Opus employ *per-application overlays* to disseminate their own service data and metadata among individual replica sites.

Clearly, a primary concern is ensuring the scalability and reliability of the service overlay. Opus addresses this through the aggressive use of hierarchy, aggregation, and approximation in creating and maintaining scalable overlay structures.

3.2 The Grid

Although it initially began as a way for scientific applications to use "excess" computing cycles at other institutions, the proponents of *The Grid* have recently embraced a more general model for resource management and sharing across a federated set of suppliers, and recent work on defining an "open grid service architecture" [5] has made it clear that the eventual target is no longer limited to relatively short-lived jobs, but also embraces longer-lived services.

3.3 Planetary scale computing

Beginning with the HP Utility Data Center [4], a product to enable the deployment of a first form of managed utility computing, HP has entered on a path to develop technology to enable what they call "planetary scale computing," or "service-centric computing" – essentially the vision espoused here. Here, the data center runs a "utility OS" [8], whose dependability is crucial to the availability of services that the data center supports. Such an "OS" has to deal with all the usual issues: resource management, provision of abstractions, client isolation ... except that the resources are entire processor nodes, or portions of disk arrays, and shared networking infrastructure, rather than the more traditional memory pages, CPUs, and IO cards.

Existing HP research work on automatic management of storage system services has demonstrated that the "lights out" provisioning of resources to meet application needs is a viable approach [1]; the next step is to apply these ideas to the broader scope of the entire data center.

4 Defining dependability

Implicit in this whole discussion is an underlying notion of what "dependability" means. Today's storage vendors and web server hosting services often use percentage uptime (e.g., 99.99%) to describe system dependability. This is a simple availability metric – "is it up?" – which, although somewhat useful for a single computer, such as a client, is inadequate when the larger context is considered,

because failures often degrade service rather than fully interrupt it.

A better notion is *performability*, which we define to mean “what portion of the time is the system meeting [the user’s] expected service levels?” Given such a definition, we can start to judge alternative service designs and offerings, and then go on to design a service deployment against its user expectations.

A service is useful only if a user’s requests can be processed within their *tolerance*, or expectation. The tolerance can include a rich combination of aspects, including throughput, latency, accuracy, completeness, and consistency (e.g., the service may return “slightly” inconsistent [9] data in exchange for improved overall accessibility).

Inadequate performance may result from many causes: network congestion, server overload, partial failures of resources, or partial data inaccessibility (or even loss); or even simply stringent user expectations. A service may be “unavailable” from a particular user’s perspective even when the system is up and running - and this is a particular problem during times of peak demand, which are precisely the times when the system needs to be most dependable.

Interruptions may be frequent and short, or rare and long. Do these have the same “average” dependability? This depends on what the user expectation is. For example, if the interruptions are frequent enough to prevent them completing a transaction, then they are unlikely to be satisfied, whatever the “average” may indicate.

Our approach to building dependable systems has applications specifying the relative value (“utility”) of various levels of performability and data consistency. A specific example of this kind of service specification for the storage-systems space can be found in [7]. Explicit in this proposal is the notion that there may be more than one appropriate service level, and that the traditional “all or nothing” distinction may not be sufficient – “is it an acceptable service?” is a more sophisticated question than “is it up?”.

In this manner, the compute utility can determine how to provision available resources to maximize per-service dependability in the face of individual failures, changing network conditions, and dynamic client access patterns.

5 Conclusions

This paper takes somewhat of a contrarian position on the question of how to build a dependable operating system. We believe that the traditional operating system, defined as a monolithic structure mediating all application access to host software, is becoming less and less important as a determiner of dependability. Rather, the “operating system” is being extended to cover the gamut of management and deployment issues involved in executing an entire service, across the network [8].

Thus, we believe that the operating system research agenda must address issues that encompass the concerns raised by such global scale resource-management: how should the “operating system” best manage global network resources to deliver reliable services transparently to millions of simultaneous users? How should it dynamically place functionality and employ redundancy to deliver much better performance and availability than any centralized host or single client system could? Dependable computing is not (just) about building a more robust UNIX or Windows. Rather, it is about thin, stateless, disposable clients utilizing dependable communication to access global, dependable, service utilities.

References

- [1] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running Circles Around Storage Administration. In *Conference on File and Storage Technology (FAST’02)*, January 2002.
- [2] Rebecca Braynard, Dejan Kostić, Adolfo Rodriguez, Jeffrey Chase, and Amin Vahdat. Opus: an Overlay Peer Utility Service. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, June 2002.
- [3] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, October 2001.
- [4] Hewlett Packard Corporation. Utility Data Center. www.hp.com/solutions1/infrastructure/solutions/utilitydata/overview/, 2001.
- [5] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002.
- [6] Patrick Goldsack. SmartFrog: a framework for configuration. In *Large Scale System Configuration Workshop*, November 2001.
- [7] John Wilkes. Traveling to Rome: QoS Specifications for Automated Storage System Management. In *International Workshop on Quality of Service (IWQoS’2001)*, June 2001.
- [8] John Wilkes, Patrick Goldsack, G. (John) Janakiraman, Lance Russell, Sharad Singhal, and Andrew Thomas. eOS - The Dawn of the Resource Economy. Technical report, HP Laboratories, May 2001. Available from <http://www.hpl.hp.com/SSP/papers>.
- [9] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, October 2000.
- [10] Haifeng Yu and Amin Vahdat. Minimal Replication Cost for Availability. Technical report, Duke University, January 2002. Submitted for publication.



**Dependency on O.S. in long-term programs:
Experience report in space programs**

Patrick CORMERY

Tel: + 33 (0) 1.39.06.25.11

Fax: + 33 (0) 1.39.06.27.97

patrick.cormery@launchers.eads.net

Le Vinh Quy RIBAL

Tel: +33 (0) 1.39.06.29.07

Fax: +33 (0) 1.39.06.27.97

le-vinh-quy.ribal@ launchers.eads.net

Arnaud STRANSKY

Tel: + 33 (0) 1.39.06.32.18

Fax: + 33 (0) 1.39.06.27.97

arnaud.stransky@ launchers.eads.net

Abstract

Space programs like launchers are typical of long-term programs: the duration may exceed 15 years. On the other hand software components like Operating Systems evolve more quickly, about 4 to 5 years.

Illustrated in the particular case of EADS-LV space programs, with real time embedded software developed in Ada language, this paper gives an overview of:

- Dependency of an embedded software on Operating System
- The relatively fast evolution of technology, languages and standards
- Industrial solutions to keep the OS permanence during more than 10 years.

1. INTRODUCTION

This paper presents the main dependencies that are encountered in long-term programs, in the particular case of space launchers programs, with Flight Programs developed in Ada language.

Space programs are typical of long-term programs. The complexity of a space system, like a launcher, needs a deployment of competencies in many different fields, and some large industrial facilities. This kind of projects last generally more than 10 years, and sometimes beyond. For example:

- ATV (Aircraft Transport Vehicle): start of development in 1998, up to 2013: 15 years.
- Ariane 4: start of development in 1982, launches up to 2003: 21 years.
- Ariane 5: start of development in 1989, launches expected up to 2010: 21 years.

Therefore the critical components used, as the Operating System of the On-Board Flight Program, shall grant permanence for this duration (permanence of the product and support services such as maintenance, hot line...).

Moreover in launchers production multiple units of a launcher are produced during the project, and they evolve continuously. A target, like the On-Board Computer or the architecture of the Flight Program may keep the same but some features like algorithms or equipments may evolve. Therefore this requires maintaining knowledge and competency in the embedded software development, even for more than 15 years.

2. DEPENDENCY ON OPERATING SYSTEM

An Operating System provides services for real-time features: multi-tasking management, static and dynamic memory allocation, interrupt management... This is an essential part of a Launcher Flight Program, that shall manage cyclic and acyclic commands, parallel computing...

The main dependencies on an OS are:

- Ascendant compatibility between standards (Ada rendezvous behaviour, ...),
- Specific real-time extension features (semaphores, ...) which are included in the provided OS,
- Target that support the OS (microprocessor, family...) and host workstation for the Software Development Environment,
- Maintain of knowledge and competencies of the OS and user's development teams.

2.1 SEMANTIC FEATURES

In the case of an advanced language as Ada, specialised in multitasking and concurrent systems, the Operating System is nearly close to the language semantic. To keep the Ada standard semantic, the OS shall include an Ada runtime compliant to the standard.

Different ways may be chosen to support a multitasking Ada software:

- Ada software on an Ada OS: the most adequate method.
- Ada software on commercial OS, with inter-layer Ada runtime: this method allows using commercial software, but need the development of inter-layer software that may decrease the performances and the behaviour of the user software.
- Ada software on commercial OS, with procedural calls: in this case the runtime is not fully compliant to the Ada standards, and the software shall call OS-specific functions. This solution is the most dependent on a given OS

2.2 TARGET AND HOST WORKSTATION

Operating Systems are generally a part of a full package that includes OS, compiler, debugging tools, etc. This environment is dedicated to a set of microprocessor or microprocessor families. Experience shows that if software are in theory compliant for a given family (example: transputters, sparcs...), there are still incompatibilities in practice.

2.3 HUMAN RESOURCES AND KNOWLEDGE MAINTAIN

One last point to take into account is the knowledge of developer teams in some specific languages or Operating Systems. The staff training is a large part of a company life, and maintaining an old language or a product specific knowledge may affect a large part of the costs.

3. OPERATING SYSTEMS AND LANGUAGE EVOLUTION

In the case of Ada language, at least three different standards can be identified currently:

- Ada 83: the standard was established in 1983.
- Ada 95: an enhancement of the standard Ada 83 (12 years after Ada83). The Ada 95 is in theory fully compliant with the Ada 83 standard, with ascendant compatibility. The Ada 95 standard introduce among others the object-oriented programming ("*protected objects*").
- Ravenscar profile: a subset of the Ada 95 standard, established in 1999 (4 years after Ada95). This profile is a set of restrictions of the Ada 95 standard, to be more determinist, and to facilitate the certification of a system based on this profile. This profile restricts in particular the Ada 83 rendezvous, concurrent with protected object behaviour.
- Ada 0Y: a Ada95 new amendment is currently (3 years after Ravenscar) studied by THE AFNOR (French standardisation organisation).

As Operating Systems comply with one or another standard, some semantics aspects may not be compliant between two different OS, despite these OS are all called "Ada compliant".

For example, the rendezvous semantic, a typical Ada features to synchronise tasks each together at precise points of execution is implemented by specific calls to task entries in Ada83. In Ada95 this mechanism is replaced by protected object, with suspending of a task on a protected flag include in the object. As in Ada95 the Ada83 rendezvous mechanism was kept, for ascendant compatibility, this was removed from the Ravenscar profile.

Therefore a Ravenscar-compliant OS is not fully compliant to Ada83 software, some changes have to be made to replace the rendezvous behaviour at least.

Moreover an OS may evolve on account of the OS provider. Reasons of change may be:

- bug correction,
- new requirements from other customers on a product,
- compiler change,
- provider commercial strategy: to change of technology, change of market...

So the dependency is a critical point to manage.

4. SOLUTION ON DEPENDENCY PROBLEM

The Operating Systems providers propose two main policies to preserve product knowledge:

- The freeze policy: the provider engages itself to keep the product at a given version, with documentation and development tools (software and hardware, like host workstations), during the freeze duration. In case of bug, the provider has the adequate equipment and shall be able to correct the bug.
- The knowledge repurchase: after a period of maintain by the provider, it can propose to the customer to purchase the rights on a product, like an Operating System, with source code, documentation, and rights of modification. The maintenance and bug correction became under the responsibility of the industrial customer. As this method grants that the product shall live as long as the project, this requires that the industrial customer develops and maintains a knowledge in some specific field like Operating System development.

If one of these mechanisms wasn't planned at the beginning of a program, and in case of change of a hardware part like the microprocessor, the industrial can ask a provider to redevelop a specific Operating System with the same interfaces than the previous one (in term of semantic and performances). Of course, this method has a cost impact higher than a commercial product.

5. CONCLUSION

This specific sort of application: space and long-term program, lead to a contradiction between the need to freeze the tools, and the evolution of technologies. Industrials have generally to choose some large and stable companies as OS providers to ensure the products permanence. This is to the detriment of the new technologies evolution.

At the beginning of such programs, industrials have to anticipate the future product and technology. Moreover companies may participate to standardisation working groups, to anticipate standards and specify some industrial requirements.

Design and Implementation of the Lambda μ -Kernel based Operating System for Embedded Systems

Kenji Hisazumi
Tsuneo Nakanishi

Teruaki Kitasuka
Akira Fukuda

Graduate School of Information Science and Electrical Engineering, Kyushu University
6-1 Kasugakouen Kasuga-city, Fukuoka 816-0922, Japan.
{nel, kitasuka, tun, fukuda}@f.csce.kyushu-u.ac.jp

Abstract

With large-scale embedded systems, improvement of development efficiency is one of the most important problems. In this paper, we design and implement an embedded operating system, called the Lambda operating system, which improves the maintainability and development efficiency of the operating system. The Lambda operating system employs micro-kernel architecture, which allows the operating system to be easily designed. In addition, we propose a method to improve operating system performance by reconstructing it in implementation. With the method, Lambda is implemented as monolithic kernel. The method allows the operating system to be quickly developed and gives high performance. This paper also shows that the method is useful through implementing a prototype of Lambda and its performance evaluation.

1. Introduction

Since portable telephones and information electric home appliances come onto the market, embedded systems become more complex. Recent embedded systems have to provide functions such as multimedia data handling and network one. The development term of the large-scale embedded systems increases more than before. On the other hand, short-term development and time-to-market are required due to severe competition. Therefore, improvement of development efficiency is one of the most important issues.

Although maintainability and development efficiency of

embedded operating systems are important issues, there are very few studies on them.

In this paper, we design and implement an embedded operating system, called the Lambda operating system, which improves the maintainability and development efficiency of the operating system.

Typically the maintainability, development efficiency and performance are exclusive. For example, an operating system which employs μ -kernel structure is easy to develop. However performance of this one is bad. Most of embedded systems require high performance. On the other hand, an operating system which employs monolithic structure is faster than μ -kernel structure. This structure needs careful developing to keep maintainability and development efficiency.

Therefore we propose transforming technique μ -kernel structure to monolithic one to concomitants with high performance and maintainability.

2. Memory protections for embedded software

Most of embedded operating systems don't have any protection to keep the constraint of memory size, performance and so on. However memory protections become important since scale of software for embedded systems is becoming large. In this section, we consider the purpose of memory protections for embedded software.

There are some purposes of memory protections.

- For protection of text and data.
To protect text and data from invalid memory access at runtime is necessary for the system to prevent into fatal.

- For security.
We download software into embedded systems from the Internet. This software may destroy the embedded system software.
- For isolating untrusted software.
When the developer buy an application program from other company (ex. WWW browser), this program may be unstable. This case is to protect other software from untrusted software.
- For debugging.
Developing software may destroy other stable software area. If there is no memory protection, you cannot find this invalid memory access. If there is it, operating systems tell us it. To remove this protection after development is better.

The protection for debugging is especially interesting. Most of programs in embedded system are not changed after debug and we can trust them. In developing phase, we run programs under protection for debug. After development we remove protection of this part and get high performance. We consider of technique of removing protections next section.

3. Lambda Operating System

The Lambda operating system employs μ -kernel architecture, which allows the operating system to be easily designed. Embedded systems have various hardwares and we must develop device drivers for them. This feature is very important for embedded systems. However, μ -kernel architecture is slower and consumes more memory than monolithic architecture. Although these weak points are improved by L4[5] roughly, but it is not enough. Cost and performance of embedded systems is more important than general-purpose systems. Therefore the Lambda operating system improves performance more, and cope with both performance and easy development by two features: the selectable memory protection and the automatic change from μ -kernel structure to monolithic structure.

The first feature of the Lambda operating system is the selectable memory protection. The Lambda operating system provides a mechanism that allows processes of system servers and application programs to be implemented in either user process mode or kernel mode. In user mode, processes run with memory protection. In kernel mode, they run without it. At development phase, processes run in user

process mode to debug the system. In this phase, we can use memory protection for effectively debugging. At final phase, memory protection can be removed. The system functions run in kernel mode. In this phase, system performance is higher than that in development phase.

Second feature of the Lambda operating system is the change from μ -kernel structure to monolithic structure. A thread, which runs in kernel mode still needs inter-thread communications (ITC) to call other thread functions. Therefore system performance with this implementation is lower than that with monolithic structured implementation. The Lambda operating system alleviates the ITC overheads by automatically replacing the ITC mechanism to the function call mechanism. System performance with the function calls mechanism is higher than that with the ITC mechanism.

4. Changing μ -kernel structure to monolithic structure

Now we consider a changing method from μ -kernel structure to monolithic one. This method can be applied to only Remote Procedure Call (RPC). The Lambda operating system often uses RPC. RPC is generally used with small codes called stub. When a client calls a client stub, the client stub encodes arguments and sends a request to a server using RPC. A server stub receives the request, decodes arguments, calls a server function, and processes the request.

We implement a new stub generator called Lambda Interface Generator (LIG). It generates stubs and templates. If the server is in the same process of the client, the call from the client is performed by the function call. On the other hand, when the callee server is in a different process from the process of the client, its call is performed by using RPC rather than function calls (Figure 1). This method reduces communication overheads between a client and a server both of which are in the same process by replacing RPC to function calls.

5 Implementation and Evaluation

We implement a prototype of Lambda on a target machine of a Celeron 300A processor. We measure the execution time of some various models. Table 1 shows response times of basic system calls for reference. Figure 2 shows measurement models of same process RPC(a), cross process RPC(b), same process RPC with the changing method

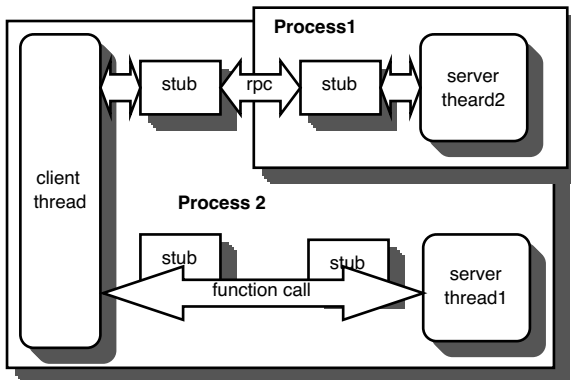


Figure 1. Server stubs and client stubs

Table 1. Response time of basic system calls

system call	cycles
thread_create	3630
process_create	30000
thread_sched	intra process 411
	inter process 1986

of Lambda described before(called monolithic RPC) (c). Figure 3 shows the execution times of each model. The result of this measurement shows that the transformation of process structure improves RPC performance 20% and the changing method improves the system performance about 10 times.

6. Related works

LRPC[3] is a technique for reducing RPC overhead as same as our method. The LRPC reduces it dynamically. On the other hand, our method reduces it statically because of a technique for embedded systems. Our method allows to optimize globally with a compiler.

Component base operating systems, such as VEST[4] Knit[2] EPOS[1], improve descriptiveness with small object code and a good performance. However it is difficult to test a component. A component of the Lambda operating system can be run and tested with protections.

7. Conclusions

In this paper, we design and implement an embedded operating system, called the Lambda operating system. The

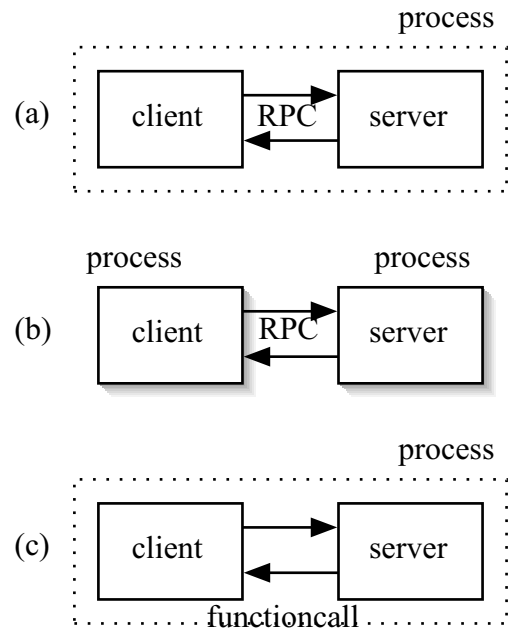


Figure 2. The structure of measurement models

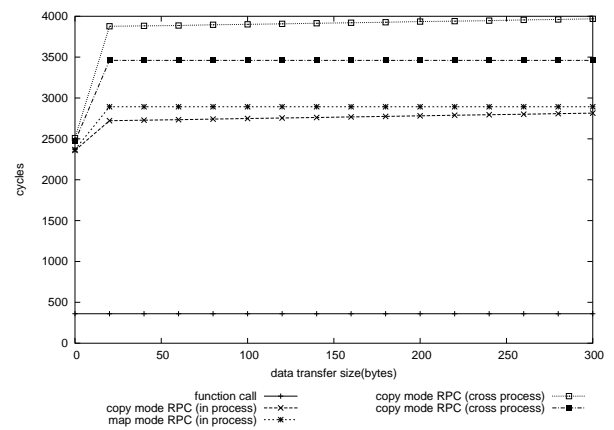


Figure 3. The performance of cross process RPC, same process RPC and monolithic RPC

Lambda operating system achieves both high maintainability and high development efficiency of the operating system. The Lambda operating system employs μ -kernel architecture, which allows the operating system to be easily designed. In addition, we propose a method to improve operating system performance by reconstructing it in implementation. With the method, Lambda is implemented as monolithic kernel. The method allows the operating system to be quickly developed and gives high performance. This paper also shows the usefulness of our method through implementing a prototype of Lambda and its performance evaluation.

Acknowledgments

This research is partly supported by Core Research for Evolutional Science and Technology (CREST) Program "Advanced Media Technology for Everyday Living" of Japan Science and Technology Corporation(JST).

References

- [1] A.A.Frohlich. Tailor-made operating systems for embedded parallel applications. In *Proc. of the 4th International Workshop on Embedded HPC Systems and Applications*, number 1586, pages 1361–1373, 1999.
- [2] A.Reid, M.Flatt, L.Stoller, J.Lepreau, and E.E.Knit. Component composition for systems software. In *In proceedings of 4th Symposium on Operating Systems Design and Implementation*, pages 347–360. Usenix Association, 2000.
- [3] B.Bershand, T.Anderson, E.Lazowska, and H.Levy. Lightweight remote procedure call. In *Proc of the 12th ACM symposium on Operating systems principles*, pages 102–103, 1989.
- [4] J.A.Stankovic. Vest: A toolset for constructing and analyzing component based operating systems for embedded and real-time systems. Technical Report No. CS-2000-19, Dept. of Computer Science, Univ. of Virginia, 2000.
- [5] J.Liedtke. Improving ipc by kernel design. In *Proc of 14th SOSF*, pages 203–205, 1993.

Efficient Heartbeats and Repair of Softstate in Decentralized Object Location and Routing Systems

Hakim Weatherspoon and John D. Kubiatowicz
 Computer Science Division
 University of California, Berkeley
 {hweather, kubitron}@cs.berkeley.edu

Abstract

Redundancy alone is not sufficient to provide long-term guarantees in distributed systems. Instead, it must be coupled with mechanisms for automatic maintenance. In this paper, we show how Decentralized Object Location and Routing networks (DOLRs) with locality provide a framework for efficient heartbeats and continuous system repair.

1. Introduction

Recent peer-to-peer systems have adopted decentralized object location and routing (DOLR) infrastructures to assist in organizing and manipulating their data. Prominent examples of DOLRs include CAN[6], Chord[8], Pastry[2], Tapestry[3, 11], and other Plaxton, Rajaraman, Richa[5] structures. DOLRs provide sufficient probabilistic routing guarantees to find an object if it exists; but not enough, if any, reliability guarantees. This dilemma is often solved with replication[1, 2] or other forms of redundancy[4, 9]. Unfortunately, redundancy is a short-term mechanism since hardware eventually fails, software has bugs, people make mistakes, and regions of the infrastructure may be destroyed by natural disasters or malicious attacks. Thus, it is essential that systems provide long-term maintenance through automatic *fault detection* and *repair* of faults.

Fault detection and repair are significant challenges in global-scale DOLR-based systems since information is embodied in the aggregate resources of a constantly changing set of unreliable nodes. The sheer size of such systems dictates that each participant will possess enough storage to hold only a small piece of the system. This requires distributed maintenance techniques. Fortunately, DOLRs that provide *locality*¹ can aid in automatic maintenance by sup-

¹By locality, we mean the ability to utilize local resources over global ones whenever possible.

porting efficient heartbeats to detect faults and trigger repair of the DOLR and resident objects.

In this paper, we survey automatic fault detection and repair techniques for infrastructure state and data. We begin by specifying the properties that we need from the underlying DOLR in Section 2. In Section 3, we briefly describe how location-independent routing works in a DOLR in the context of Tapestry. Next, we survey some DOLR implementations in Section 4. Finally, we enumerate the properties of a DOLR that enable self-repair in Section 5.

2. Requirements

Replicas are distributed widely to enhance durability. This complicates the process of locating them for repair. A closely related problem is the need to direct *queries* to appropriate nodes. Since the *key* for routing to a *value* is named by opaque bit-strings – a *globally-unique identifier* or *GUID*, we need an infrastructure that can perform *location-independent routing* of messages directly to objects using only GUIDs. In addition, the routing layer should exhibit *deterministic location*, objects should be located if they exist anywhere in the network.

Location-independent routing in DOLR's use *softstate* to allow the system to be dynamic. DOLR's maintain softstate with *heartbeats* and/or republish messages (*i.e.* re-add an entry into the distributed directory). Server heartbeats help maintain routing state. Object heartbeats help maintain the distributed directory. Heartbeats assist in detecting failures. When a failure occurs, the system needs to account for the loss and refresh loss redundancy if a *threshold* is reached.

Problem If not careful, the cost for maintaining the routing structure and objects stored in it can easily render the system useless. The *maintenance resource over utilization* problem has three aspects that are reflected in the current literature of DOLR's: server heartbeats that cross the wide area, object heartbeats on a per-object-basis, and use of re-

dundant links for heartbeats and maintenance information. Per-object heartbeats can be considered dangerous if the system scale becomes large enough.

Heartbeat Locality If the the overlay network (DOLR) is not aware of the underlying network topology, server and object heartbeats will cross the wide area increasing the system's bisection bandwidth utilization. To permit locality optimizations it is important that the routing process exhibit *routing locality*, use as few network hops as possible and that these hops should be as short as possible. That is, routes should have low *stretch*², not just a small number of application-level hops.

Per-Object Heartbeats Each node in a DOLR has the potential to store many objects; that is, more objects than neighbor links. Some of the DOLR's make no provisions to make sure that objects still exist; while, other DOLR's detect object failure with object heartbeats.

Redundant Links The problem with object heartbeats is that they traverse redundant links. That is, the number of stored objects is much greater than the number of neighbor links. Most of the heartbeats can be aggregated together.

3. Location-Independent Routing

We now describe Tapestry[3, 11], a routing and location system. Tapestry is an IP overlay network that uses a distributed, fault-tolerant architecture to track the location of every object in the network. Tapestry has two components: a *routing mesh* and a *distributed directory service*.

Routing Mesh: Figure 1 shows a portion of Tapestry. Each storage server and client is a Tapestry node with a unique 40-digit hexadecimal address drawn from a random distribution. Tapestry nodes are connected via *neighbor links* of varying levels; these are shown as solid arrows. The level-1 links (L1) from a given node connect to the 16 closest, defined by network latency, nodes with different values in the lowest digit of the address. Level-2 links (L2) connect to the 16 closest nodes that match in the lowest digit and have different second digits, *etc.*

Neighbor links provide a route from every node to every other node. For example, Figure 1 shows a path (thick solid arrows) from node 5230 to node 8954. The routing process resolves the destination one digit at a time: $8*** \Rightarrow 89** \Rightarrow 895* \Rightarrow 8954$, where *'s represent wildcards. This scheme is based on the hashed-prefix

²Stretch is the ratio between the distance traveled by a query to an object and the minimal distance from the query origin to the object.

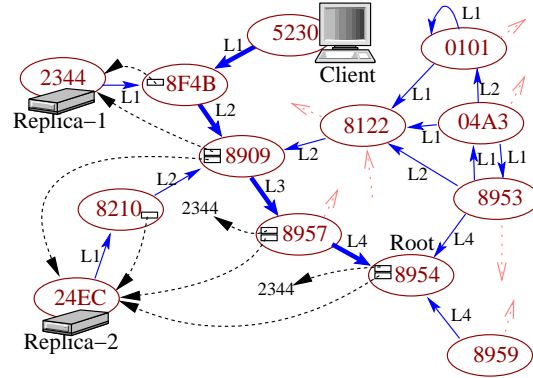


Figure 1. Tapestry Infrastructure: Nodes are connected to other nodes via neighbor links (solid arrows). Any node can route to any other by resolving one digit at a time: e.g. 0101 → 8122 → 8908 → 8957 → 8954. Each GUID is associated with one particular “Root” node (8954 in this example). A server publishes the location of a replica by sending a message toward the root, leaving back-pointers at each hop (dotted arrows). Clients locate replicas by sending a message toward a root until they encounter enough pointers. Client 5230 can locate two replicas after only two hops: 5230 → 8F4B → 8909.

routing structure originally presented by Plaxton, Rajaraman, and Richa [5].

Distributed Directory Service: To perform location-independent routing, Tapestry employs a mechanism that deterministically maps each GUID to a small (5) set of unique *root* nodes. A storage server then *publishes* the fact that it is storing a replica or other object by routing a message toward each of the root nodes (as described above), depositing *location pointers* to the object's location at each hop from the server to the root. Figure 1 shows two replicas with the same GUID stored at different nodes (nodes 2344 and 24EC). This figure also shows one of the root nodes (8954). The location pointers are shown as dotted arrows that point back to storage servers. Note that each of the root nodes keeps track of the location of every replica that maps to its address, enabling distributed repair (Section 5).

To locate an object, a client sends a message toward one of the object's roots. As soon as the message encounters a pointer with the desired GUID, it routes directly to the object. In the figure, client 5230 can locate two replicas after only two hops: 5230 → 8F4B → 8909. In the worst case, this involves routing all the way to the root. However, if the desired object is close to the client, then the path from the client to the root will intersect the path from a storage server to the root with high probability. In fact, it is shown

Scheme	Maintain Node	Maintain Object	Detect Node	Detect Object	Repair Object
PRR [5]	-	-	-	-	-
Chord [8]	server hb	-	timeout, msg	-	-
CAN [6]	server hb	-	timeout, msg	-	-
Pastry [7]	server hb	-	timeout, msg	-	-
Tapestry [3, 11]	server hb	obj hb republish	timeout, msg	timeout, msg	-
Tapestry w/ This paper	server hb expn bkf	server hb expn bkf republish notification	timeout msg	timeout msg	threshold

Table 1. *Repair in DOLR systems: Most systems implement server heartbeats (hb) to maintain routing state and detect node failure. Similarly, objects are maintained with object heartbeats or explicit republish messages. Object Failure is detected when a timeout occurs or a republish has not been received. Finally, Lost objects are repaired when a redundancy threshold is reached.*

in [5] that the average distance traveled in locating an object is proportional to the distance from that object³.

4. System Comparisons

Table 1 compares maintenance techniques for several DOLR networks. All DOLRs can locate objects. However, they differ with respect to locality, *i.e.* not all systems minimize stretch. PRR [5], Tapestry[3, 11], and Pastry [7] provide locality in the connections between nodes in the DOLR, assisting efficient node-level heartbeats. CAN [6] and Chord [8] do not have such locality by default, but may evolve local connections over time. PRR and Tapestry also provide minimal stretch in routing to objects, providing for efficient object heartbeats. Pastry, CAN, and Chord have heuristics to reduce object location cost, so they may perform well in practice; however, efficient object-level heartbeats may be difficult to construct.

All the systems (except PRR, which assumes a static network) use some form of server heartbeats to maintain routing softstate. The server heartbeats allow the system to detect failed nodes and possibly route around them. However, the systems differ in maintaining object state. Only Tapestry and Wells describe object heartbeat techniques and only Wells explains how to repair lost redundancy.

The problem with straightforward object heartbeats in Tapestry[11] is that with enough objects in the system, this can be quite expensive[10]; that is, a significant (over 20%) amount of a servers bandwidth resources were used for object heartbeats. Wells[10] extended Tapestry heartbeats and proposed a server heartbeat exponential backoff, critical ob-

³Experiments show a small constant of proportionality; See [11].

ject notification, and a trust metric to make maintenance, detection, and repair feasible.

Looking at this issue more carefully, however, we note that heartbeats for objects serve two totally different purposes: first, they allow us to detect that a *server* has failed and second, they permit us to repair inconsistencies in the DOLR data structures pointing at objects. The important insight is that slight inconsistencies in the DOLR can be handled through redundancy, while server failure should be noticed more precisely. We extend the work of Wells by aggregating information that will travel over the same links (multicast) to efficiently notice server failure and trigger object reconstruction. Inconsistencies in the DOLR state will still be repaired through periodically republishing object locations, but this process can be done more lazily and locally.

5. Maintenance and Repair

In a dynamic environment systems must adapt to changes in the infrastructure and repair damaged replicas or loss redundancy. A basic assumption of many DOLRs is that there are faulty and malicious nodes that attempt to corrupt data and deny service; however, we assume that there is a large number of “good” servers that properly adhere to the DOLR protocols. We also assume that nodes make provisions to keep local storage and state as stable as possible.

5.1. Infrastructure Repair

Repair needs the DOLR to adjust to changing network configurations. An example DOLR node insertion and deletion algorithm can be found in [3, 11]. Most DOLRs provide mechanisms for both planned and unplanned server removal. When possible, the departing server proactively informs its neighbors of its imminent departure so that neighbors may remove the node from their neighbor maps. Additionally, the node may move replicas to nearby nodes.

To address the unexpected departure of nodes from the network, DOLRs rely on server heartbeats. These server heartbeats are sent along neighbor pointers (as described in Section 3); the fanout of these heartbeats is limited, since the number of neighbor pointers is fixed. However, since the average network distance of these pointers increases geometrically with level number, we send heartbeats along level-1 links more frequently than level-2 links, *etc.*. By monitoring these heartbeats, the routing infrastructure can detect a server’s departure and trigger the modification of the routing mesh and redistribution of pointers⁴.

⁴An ongoing area of research is determining legitimate server failure from denial-of-service attacks.

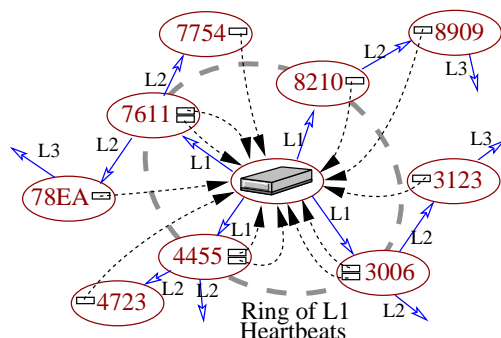


Figure 2. Data-Driven Server Heartbeats: A slice of tapestry around a server (middle node). Pointers from DOLR back to server define natural multicast tree from server to nodes containing pointers to server. This tree is used for data-driven heartbeats. Locality is achieved by traversing top levels of tree more frequently than bottom levels, and by traversing only portions of the tree for objects below a certain redundancy threshold.

5.2. Distributed Repair

Distributed repair mechanisms exploit DOLRs distributed information and locality properties⁵. Figure 2 illustrates how the pointers in a DOLR aid in detecting failure by directing server heartbeats. This figure illustrates the natural multicast tree from each server to the set of nodes containing pointers to that server. When a server crashes, this set of nodes must eventually be informed (to clean up pointers and possibly trigger repair). Repair utilizes this multicast tree for *data-driven server heartbeats*. To avoid flooding the network with an excessive number of heartbeats, we provide the same sort of exponential backoff as mentioned with the server heartbeats. Heartbeats go to the first level of the tree most frequently (shown in the figure as the “Ring of L1 Heartbeats”), second level less frequently, *etc.* With this scheme, nodes near a replicas recognize the majority of replica failures and tend to recognize them quickly, while nodes farther away protect against regional outages.

Repair employs two mechanisms to (1) keep the DOLR pointers as up-to-date as possible and (2) trigger repair as soon as the number of surviving replicas from a given block falls below a *threshold*. As shown in Figure 1, Tapestry contains information about the state of an objects replicas. In particular, the *root* nodes associated with an objects replicas know the number and location of all surviving replicas. Although this information is somewhat imprecise, it does provide a framework around which to trigger repair. When a server ceases to send its heartbeats for a sufficiently

long time, the nodes along this multicast tree recognize the failure and propagate pointer changes toward replica roots, which trigger repair when necessary.

The loss of a region of servers may require time to notice. Consequently, we can enhance the observability of certain replicas that belong to objects with “dangerously low” levels of redundancy: we inform all of the pointers on the path from server to root that we want notification as soon as the server ceases to function. The important observation here is that DOLR pointers provide a distributed framework for adjusting the rate of repair and observation as necessary.

6. Conclusion

We have discussed various distributed maintenance, detection, and repair techniques that increase system reliability. Reliable distributed systems are dependent upon location-independent properties of DOLRs. The system reliability is predicated on the use of efficient heartbeats.

References

- [1] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, October 2001.
- [2] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP*, 2001.
- [3] K. Hildrum, J. Kubiawicz, S. Rao, and B. Zhao. Distributed data location in a dynamic network. In *Proc. of ACM SPAA*, 2002.
- [4] J. Kubiawicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*. ACM, 2000.
- [5] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of ACM SPAA*, June 1997.
- [6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of SIGCOMM*. ACM, August 2001.
- [7] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, November 2001.
- [8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM*. ACM, August 2001.
- [9] H. Weatherspoon and J. Kubiawicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, 2002.
- [10] C. Wells. The oceanstore archive: Goals, structures, and self-repair. Master’s thesis, University of California, Berkeley, May 2001.
- [11] B. Zhao, A. Joseph, and J. Kubiawicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, 2001.

⁵Tapestry’s neighbor links encode network locality.

Event-driven Programming for Robust Software

Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, Robert Morris

MIT Laboratory for Computer Science

{fdabek,kolya,kaashoek,rtm}@lcs.mit.edu, dm@scs.cs.nyu.edu

Abstract

Events are a better means of managing I/O concurrency in server software than threads: events help avoid bugs caused by the unnecessary CPU concurrency introduced by threads. Event-based programs also tend to have more stable performance under heavy load than threaded programs. We argue that our libasync non-blocking I/O library makes event-based programming convenient and evaluate extensions to the library that allow event-based programs to take advantage of multi-processors. We conclude that events provide all the benefits of threads, with substantially less complexity; the result is more robust software.

1 Introduction

The debate over whether threads or events are best suited to systems software has been raging for decades. The central question of this debate is whether threads or events should be used to manage concurrent I/O. Many prefer threads because they preserve the appearance of serial programming and can take advantage of multi-processor hardware. Programmers find programming with threads difficult, however, and as a result produce buggy software. This paper contributes to the debate by showing that event-based programming can provide a convenient programming model, that it is naturally robust, and that it can also be extended to take advantage of multi-processors. For these reasons, we argue that there is no reason to use threads for managing concurrent I/O in system programs—events provide all the benefits that threads provide, but in a more robust way.

Thread-based programs use multiple threads of control within a single program in a single address space [3]. Threaded programs achieve I/O concurrency by suspending a thread blocked on I/O and resuming execution in a different thread. Under this model, the programmer must carefully protect shared data structures with locks and use condition variables to coordinate the execution of threads.

Event-based programs are organized around the process-

ing of events. When a program cannot complete an operation immediately because it has to wait for an *event* (e.g., the arrival of a packet or the completion of a disk transfer), it registers a *callback*—a function that will be invoked when the event occurs. Event-based programs are typically driven by a loop that polls for events and executes the appropriate callback when the event occurs. A callback executes invisibly until it hits a blocking operation, at which point it registers a new callback and then returns.

In this position paper, we show that writing programs in the event-based model can be convenient and show how to extend the event-based model to exploit multi-processors in a way that requires programmers to make only minor changes to their code. This multi-processor event model makes it easy to capture the levels of CPU concurrency typically available to systems programs, but does not require the programmer to insert locks or condition variables. The multi-processor support is part of the *libasync* asynchronous programming library. Unlike programming with threads, *libasync* doesn't force programmers to manage thread synchronization, doesn't force programmers to guess how much space to reserve for stacks, provides high performance under high load, and doesn't have hidden performance costs. As a result, programs written with *libasync* tend to be robust. Based on our experience with the library, we believe there is no reason to use threads for managing concurrent I/O.

2 Threads lead to unreliable software

The main advantage of threads is that they allow the programmer to overlap I/O and computation while preserving the appearance of a serial programming model. The main disadvantage of threads is that they introduce concurrent execution even where it is not needed. This concurrency unnecessarily forces programmers to cope with synchronization between threads. In practice, threaded programs almost always have latent data races and deadlocks and, as a result, they are not robust. In our view, programmers must be spared the complexities of concurrency to make software more robust.

The problems with thread-based programming have long been recognized. Ousterhout argues that the convenience of threads is not worth the errors they encourage, except for multi-processor code [11]. Engler *et al.* demonstrate that synchronization errors, particularly potential deadlocks, are common in the Linux kernel [5]. Savage *et al.* [13] found races in both student code and production servers.

While it might seem that one could eliminate synchronization by moving to non-pre-emptive threads on a uniprocessor, errors will still arise if a thread blocks (yields the CPU) unexpectedly. In complex systems, it is easy for programmers of one module not to understand the exact blocking behavior of code in other modules. Thus, one can easily call a blocking function within a critical section without realizing that pre-emption may occur [1]. Deadlocks are also a possibility when using non-preemptive threads, as one still needs to hold locks across known yields. Adya *et al.* propose the use of non-preemptive threads augmented with runtime checks to detect unexpected yields in submodules [2] and describe a new taxonomy for I/O concurrency techniques. They also show how to integrate code written in the event-driven style (described in their taxonomy as cooperative task management and manual stack management) with non-preemptive threads (cooperative task management and automatic stack management) in the same program.

Another complication of threads is the need to predict the maximum size of a stack. Most systems conservatively use one or two pages of memory per thread stack—far more than the few hundred bytes typically required per callback. The designers of the Mach kernel found stack memory overhead so high that they restructured the kernel to use “continuations,” essentially rewriting the kernel in an event-driven style [4]. Stack memory overhead is especially burdensome in embedded environments where memory is scarce. The standard practice of allocating full pages for thread stacks also causes additional TLB pressure and cache pressure, particularly with direct-mapped caches [4]. Cohort scheduling attempts to increase data and instruction locality in threaded programs by running related computations, organized as “stages,” consecutively [7].

Robust software must gracefully handle overload conditions. Both Pai *et al.* [12] and Welsh *et al.* [14] explore the advantages of event-driven programs under high load. Welsh demonstrates that the throughput of a simple server using kernel-supported threads degrades rapidly as many concurrent threads are required. Pai extends the traditional event-driven architecture to overcome the lack of support for non-blocking disk I/O in most UNIX implementations by using helper processes to handle disk reads. On workloads involving many concurrent clients, Pai’s event-based Web server provides higher performance than a server using kernel-based threads.

Writing programs in the event-driven style alleviates all

of the problems mentioned above. Data races are not a concern since event-based programs use a single thread of execution. Event-driven programs need only allocate the memory required to hold a callback function pointer and its arguments, not a whole thread stack; this reduces overall memory usage. In addition, these pointers and arguments can be allocated roughly contiguously, reducing TLB pressure. The livelock-like behavior of threaded servers under high load is avoided by event driven programs that queue events that cannot be serviced rather than dedicating resources to them [10].

Lauer and Needham observe that programs based on message passing (which corresponds to the event-driven model) have a dual constructed in the procedure-oriented (threaded) model (and vice versa) [8]. Based on this observation, the authors conclude that neither model is inherently preferable. The model described by Lauer and Needham does not exploit the fact that coordination is considerably simpler when using non-preemptive scheduling. As a result the authors’ conclusion neglects the advantages of the lower synchronization burden offered by the event-driven model.

While this paper focuses on user-level servers, the arguments apply to operating system kernels as well. In this realm, the event-driven architecture corresponds to a kernel driven by transient interrupts and traps. Ford *et al.* compare event-driven kernels with threaded kernels in the context of Fluke [6].

3 Making asynchronous programming easy

The most cited drawback of the event-driven model is programming difficulty. Threaded programs can be structured as a single flow of control, using standard linguistic constructs such as loops across blocking calls. Event-driven programs, in contrast, require a series of small callback functions, one for each blocking operation. Any stack-allocated variables disappear across callbacks. Thus, event-driven programs rely heavily on dynamic memory allocation and are more prone to memory errors in low-level languages such as C and C++. As an example, consider the following hypothetical asynchronous write function:

```
void awrite (int fd, char *buf, size_t size,
            void (*cb) (void *), void *arg);
```

`awrite` might return after arranging for the following to happen: as soon as the file descriptor becomes writeable, write the contents of `buf` to the descriptor, and then call `cb` with `arg`. `arg` is state to preserve across the callback—state that likely would be stack-allocated in a threaded program. A number of bugs arise with functions like `awrite`. For example, `awrite` probably assumes `buf` will remain intact until the callback, while a programmer might be

tempted to use a stack-allocated buffer. Moreover, the cast of `arg` to void pointer and back is not type safe.

The C++ non-blocking I/O library in use by the authors, *libasync*, provides several features to eliminate such memory problems. It supplies a generic reference-counted garbage collector so as to free programmers from worrying about which function is responsible for deallocating what data.

libasync also provides a type-safe method of passing state between callbacks. A heavily overloaded template function, *wrap*, allows the programmer to pass data between callbacks with function currying: *wrap* takes as arguments a function or method pointer and one or more arguments and returns a function object accepting the original function's remaining arguments. Thus, the state of an operation can be bundled as arguments to successive callbacks; the arguments are type-checked at compile time.

Finally, the library also provides classes to help deal with the complications of short I/O operations (i.e., when kernel buffers fill up and a system call such as *writew* only writes part of the data). The *suio* class can hold onto a reference counted object until the "printed" data is consumed by an *output* call.

Experience with *libasync* shows that it is easy to learn and use. We use the library day-to-day when implementing network applications. Students have used the library to complete laboratory assignments including web proxies and cryptographic file systems.

4 Multi-processor event programming

We have modified the asynchronous programming library described in Section 3 to take advantage of multi-processors. The modified library (*libasync-mp*) provides a simple but effective model for running threads on multiple CPUs, but avoids most of the synchronization complexity of threaded programming models.

Programs written with *libasync-mp* take advantage of multi-processors with a simple concurrency mechanism: each callback is assigned a *color* by the programmer, and the system guarantees that no two callbacks with the same color will run concurrently. Because callbacks are assigned a default color, *libasync-mp* is backwards compatible with existing event-based applications. This allows the programmer to incrementally add parallelism to an application by focusing on just the event callbacks that are likely to benefit from parallel execution. In contrast, typical uses of threads involve parallelizing the entire computation, by (for example) creating one server thread per client; this means that all mutable non-thread-private data must be synchronized. The concurrency control model provided by *libasync-mp* also avoids deadlocks: a callback has only one color, so cycles can't arise; even if multi-color callbacks were allowed,

the colors are pre-declared, so deadlock avoidance would be easy.

The *libasync-mp* model is more restrictive than many thread synchronization models; for example, there is currently no concept of a read-only color. However, our experience suggests that the model is sufficient to obtain about as much parallel speedup as could typically be obtained from threads.

libasync-mp maintains a single queue of pending callbacks. The library uses one kernel thread per CPU to execute callbacks. Each kernel thread repeatedly dequeues a callback that is eligible to run under the color constraints and executes it. An additional callback, inserted by the library, calls the `select ()` system call to add new callbacks to the queue when the events they correspond to occur.

Ideal support for multi-processor programming would make it easy to port existing programs to a multi-processor, and would make it easy to modify programs to get good parallel speedup. Thus we are interested in two metrics: performance and ease of programming. We evaluate the two criteria in an example application: the SFS file server [9].

Our performance results were obtained on a 700 MHz 4-processor Pentium III Xeon system running Linux kernel 2.4.18. Processor scaling results were obtained by completely disabling all but a certain number of processors on the server while running the benchmark.

All communication between the SFS server and clients is encrypted using a symmetric stream cipher, and authenticated with a keyed cryptographic hash. Because of its use of encryption, the SFS server is compute-bound under heavy workloads and therefore we expect that by using *libasync-mp* we can extract significant multiprocessor speedup.

The modifications to the SFS server are concentrated in the code that encrypts, decrypts, and authenticates data sent to and received from the clients. The callback responsible for sending data to the client is representative of how we parallelized this server: we split the callback into three smaller callbacks. The first and last remain synchronized with the rest of the server code (i.e. have the default color), and copy data to be transmitted into and out of a per-client buffer. The second callback encrypts the data in the client buffer, and runs in parallel with other callbacks (i.e., has a different color for each client). The amount of effort required to achieve this parallel speedup was approximately 90 lines of changed code, out of a total of roughly 12,000 in the SFS server.

We measured the total throughput of the file server to all clients, in bits per second, when multiple clients read a 200 MByte file whose contents remained in the server's disk buffer cache. We repeated this experiment for different numbers of processors.

The bars labeled "libasync-mp" in Figure 1 show the per-

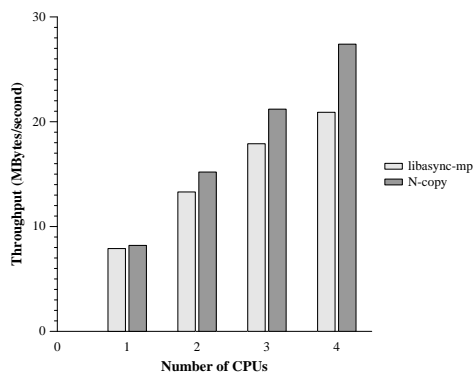


Figure 1. Performance of the SFS file server using different numbers of CPUs, relative to the performance on one CPU.

formance of the parallelized SFS server on the throughput test. On a single CPU, the parallelized server is 0.95 times as fast as the original uniprocessor server. The parallelized server is 1.62, 2.18, and 2.55 times as fast as the original uniprocessor server on two, three and four CPUs, respectively. Further parallelization of the SFS server code would allow it to incrementally take advantage of more processors.

To explore the performance limits imposed by the hardware and operating system, we also measured the total performance of multiple independent copies (as many as CPUs were available) of the original *libasync* SFS server code. In practice, such a configuration would not work unless each server were serving a distinct file system. An SFS server maintains mutable per-file-system state, such as attribute leases, that would require synchronization among the server processes. This test gives an upper bound on the performance that SFS with *libasync-mp* could achieve.

The results of this test are labeled “N-copy” in Figure 1. The SFS server implemented using *libasync-mp* closely follows the aggregate performance of multiple independent server copies for up to three CPUs. The performance difference for the 2- and 3-processor cases is due to the penalty incurred due to shared state maintained by the server, such as file lease data, user ID mapping tables, and so on.

5 Conclusions

The traditional wisdom regarding event-driven programming holds that it offers superior performance but is too difficult to program and cannot take advantage of SMP hardware. We have shown that, by using *libasync-mp*, programmers can easily write event-driven applications and take advantage of multiple processors.

References

- [1] The Amoeba reference manual: Programming guide. <http://www.cs.vu.nl/pub/amoeba/manuals/pro.pdf>.
- [2] ADYA, A., HOWELL, J., THEIMER, M., BOLOSKY, W. J., AND DOUCEUR, J. R. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proc. Usenix Technical Conference* (2002).
- [3] BIRRELL, A. D. An introduction to programming with threads. Tech. Rep. SRC 35, Digital SRC, 1989.
- [4] DRAVES, R. P., BERSHAD, B. N., RASHID, R. F., AND DEAN, R. W. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (1991), Association for Computing Machinery SIGOPS, pp. 122–136.
- [5] ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. Checking system rules using system-specific, programmer-written compiler extensions. In *Usenix Symposium on Operating Systems Design and Implementation (OSDI)* (2000).
- [6] FORD, B., HIBLER, M., LEPREAU, J., MCGRATH, R., AND TULLMANN, P. Interface and execution models in the Fluke kernel. In *Operating Systems Design and Implementation* (1999), pp. 101–115.
- [7] LARUS, J. R., AND PARKES, M. Using cohort scheduling to enhance server performance. In *Proc. Usenix Technical Conference* (2002).
- [8] LAUER, H. C., AND NEEDHAM, R. M. On the duality of operating system structures. In *Proc. Second International Symposium on Operating Systems, IRIA* (Oct. 1978). Reprinted in *Operating Systems Review*, Vol. 12, Number 2, April 1979.
- [9] MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)* (Kiawah Island, South Carolina, December 1999).
- [10] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.* 15, 3 (Aug. 1997), 217–252.
- [11] OUSTERHOUT, J. K. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX technical conference, 1996.
- [12] PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An efficient and portable web server. In *Proceedings of the 1999 Annual Usenix Technical Conference* (June 1999).
- [13] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems* 15, 4 (1997), 391–411.
- [14] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Oct. 2001), pp. 230–243.

Execution Time Limitation of Interrupt Handlers in a Java Operating System

Meik Felser, Michael Golm, Christian Wawersich, Jürgen Kleinöder
 University of Erlangen-Nürnberg
 Dept. of Computer Science 4 (Distributed Systems and Operating Systems)
 Martensstr. 1, 91058 Erlangen, Germany
 {felser, golm, wawersich, kleinoeder}@informatik.uni-erlangen.de

Abstract

Device drivers are a very critical part of every operating system. They often contain code that is executed in interrupt handlers. During the execution of interrupt handlers, the processing of some other interrupts is usually disabled. Thus errors in that code can compromise the whole system.

This paper describes an approach to ensure that an interrupt handler is not allowed to use more than a specified amount of time. Our approach is based on a Java operating system and consists of a combination of verification at compilation time and run-time checks.

1 Introduction

Modern operating systems have to support a wide range of different hardware. To be flexible the code to access the hardware is modularized in device drivers. Each driver is responsible for the communication between the operating system and the corresponding device.

The communication between hardware and software is often realized with interrupts. Hence drivers have an exceptional position because they must be able to react to interrupts. During the execution of an interrupt handler the processing of new interrupts is usually disabled. Thus an error in an interrupt handler, e.g. an endless loop, does not only affect the functionality of the driver but can have severe effects on the whole system.

Beside the worst case of an interrupt handler containing an endless loop every interrupt handler, being executed exclusively, has an influence on the response time of the system. The scheduling, in particular real-time scheduling, is reliant on the possibility to interrupt a running thread. In almost every system this is realized by the timer interrupt. But this interrupt is delayed if the processor is executing another interrupt handler.

A common approach to minimize the execution time of an interrupt handler is to split it into two parts. A first-level handler is activated directly by the interrupt. This handler performs only time-critical actions and unblocks a second-level handler which performs all other tasks. However a

faulty first-level handler can still compromise the whole system.

In this paper we introduce an approach to ensure an upper bound execution time for first-level interrupt handlers. We describe a Java bytecode verifier that is part of our Java operating system JX [GFW+02]. All device drivers for this system, including their interrupt handlers, are completely written in Java. The verifier can either predict an upper bound execution time for a method when the bytecode is translated into machine code or it extends the method to execute run-time checks.

In the next section we classify the possible errors of device drivers. After that we introduce our verifier in sections 3. In section 4 we describe the execution time analysis. Section 5 concludes the paper.

2 Classification of Errors

To estimate the probability of faulty device drivers, we take a look at the Linux kernel. More than 77% of the source code lines are device drivers (Linux 2.4.18). The large amount of supported devices implies that not every possible combination of drivers was tested. Beside implementation errors, side effects can cause unpredictable errors. Other studies e.g. [CY01] circumstantiate that most errors are found in device drivers.

For the further analysis we divide the errors in device drivers into three classes, according to the effect of their misconduct:

- Errors causing an instant system crash.
- Errors causing an instant crash of the device driver.
- Errors that do not crash the driver but lead to unexpectedly long execution.

The first class of behavior can be caused by faulty DMA transfers, for example, due to a wrong initialization of the device's DMA engine. It is very hard to counteract these effects in general, without any knowledge of the individual device. In the scope of this paper we trust the DMA initialization sequence of the driver ([GKB01]) and focus on the other classes of errors.

Implementation errors, such as buffer overflows or invalid pointers often lead to segmentation violations (or similar faults) and characterize the second class of errors. Using Java makes many of these bugs impossible because of the typesafety and the boundary checks at array accesses. But even without a typesafe language, an appropriate protection mechanism can delimit the effect of these errors, such that only the driver is compromised. After removing the driver, a new instance can be started, assuming that the error does not occur again, at least not immediately.

Errors of the third class are mainly based on wrong preconditions or caused by unexpected side effects from other device drivers. But also simple implementation faults can provoke these effects. If we do not fully trust our device drivers we must be able to handle such errors.

3 The Bytecode Verifier

When a Java class is loaded by our system, its bytecode is analyzed by a verifier and translated into machine code. The verifier performs several tasks:

- It checks whether the byte code complies with the JVM specification.
- It tries to analyze the worst-case execution time.
- It checks the applicability of some optimizations. The null pointer analysis, for example, checks whether a reference can never become null. In this case the compiler does not need to insert a run-time check.

To prevent erroneous drivers especially the first two steps are important. In the first step the verifier checks whether the bytecode fulfills the Java virtual machine specification [LY99]. This type of verification is performed by most Java run-time systems. The major task is to ensure pointer safety. It must be guaranteed that no numeric value is interpreted as reference. This is mainly realized by testing whether the number and types of operands on the stack is correct for each operation. Besides this, the access restrictions of variables and methods are checked. The compliance to the JVM specification guarantees typesafety and thus eliminates all errors related to invalid pointers. The second step is described in the next section.

4 Execution Time Limitation

In this section we describe the worst-case execution time analysis of our bytecode verifier. In contrast to other papers (e.g. [OS97], [EE00]) we are not interested in the tightest possible worst-case execution time or the exact program flow. We only want to ensure that a specified amount of time is not exceeded.

4.1 Worst-case execution time analysis

To analyze the worst-case execution time for a method the verifier builds up a flow graph of the respective method.

In [Sha89] an additive rule is proposed to evaluate the worst-case execution time of a program based on the worst-case execution time of parts e.g. single instructions. With the prerequisite, that the worst-case execution time of every Java bytecode is known, we only have to find the longest path through the flow graph.

One problem is, that the run-time estimation of some bytecodes is very hard, for example the allocation of a new object can lead to an GC run of unpredictable duration. Therefore we prohibit the GC to run in an interrupt handler and have reserved a little amount of the heap for object allocation in interrupt handlers.

The second problem is, that the program flow is rarely linear. A conditional jump (e.g. caused by an *if* statement) splits the program flow into two branches. In the case of an *if* statement we have to estimate the execution time for each branch separately and use the maximum. It is more complex if the flow graph contains backward jumps leading to cycles (e.g. caused by loops). The execution time of simple loops with known length is calculated by multiplying the execution time of the body by the number of iterations. Therefore the problem is to determine the number of iterations for each loop.

We distinguish three types of loops:

- simple loops with fixed length,
- simple loops with simple condition function, and
- complex loops.

The first type is the easiest, but very few loops are of that type. The second type can be managed in an analytical approach [Bli94]. If the loop condition depends on a monotonic function, a fixed upper bound can be evaluated. But the number of iterations mostly depend on relatively complex loop conditions. A general approach to analyze the execution time of every kind of loop is impossible without further information from the user [Par93] or basic restrictions concerning the kind of loop conditions [PK89].

Since our goal is not to evaluate the worst-case execution time, but to limit it, we do not use an analytical approach to handle loops. Instead we use an approach of partial execution. *If* statements are handled as described above. But if we recognize a loop we execute its condition function to determine whether another iteration is executed.

4.2 Partial execution

On Java bytecode level loops are realized with conditional jumps as in most machine languages. Therefore the first task is to determine which conditional jump leads to a loop and which is only caused by an *if* statement. The cor-

relation between the parameters of the loop condition and the number of iterations needs not to be analyzed. We only have to determine which parameters, more precisely which bytecodes, influence the number of iterations. These bytecodes are recognized in a recursive application of simplification rules [Alt01] which try to identify typical constructs, such as *if* conditions or simple loops. In the context of this paper we are not interested in the details of this process. We concentrate on the results. The analysis supplies us with a list of bytecodes which have to be executed to determine the length of loops. This splits the bytecodes into two categories. Those which have to be executed and those we only need to simulate.

To simulate a bytecode means that we only have to add the execution time of the operation to our sum. Whereas we need to evaluate a result for each bytecode which was marked for execution because it is needed by another operation, for example, a conditional jump. Special attention must be paid to the *invoke* bytecodes. These instructions are used to call another method. If these bytecodes are used, the verifier analyzes the called method and begins the partial execution of that method unless there already exists valid execution time information for that method (e.g. due to a prior analysis of the method).

If the execution of a bytecode depends on parameters which were defined outside the scope of the analysis, a start value for each parameter must be provided at the beginning of the partial execution. If we do not trust the source of these parameters we have to insert checks that verify the given values at run time and ensure that they match the expected values or are within an expected range of values.

The approach of partial execution is not able to estimate the worst-case execution time for all methods. For example if the method contains an endless loop, the partial execution loops infinitely. But this is not a problem since we are only interested, whether a method is executed within a specified amount of time. By checking the elapsed time during the partial execution this can be evaluated for almost every method. Thus this procedure is sufficient for our intention.

4.3 Checks at run time

Some situations prohibit the use of the partial execution approach. First it is not always possible to supply a suitable set of start values. Second the Java bytecode allows loop constructs which can not be handled by our simplification rules (e.g. if a basic block is shared by two loops), therefore the flow graph can not be reduced. Although bytecode of such complexity is never created by a Java to bytecode compiler, handwritten bytecode can contain such complex loop constructions and, anyhow, comply to the JVM specification.

In relation to interrupt handlers this is almost no problem. Most interrupt handlers are only dispatchers. Thus they have a linear control structure and the verifier can determine the worst-case execution time at compile time. Alternatively complex non-linear interrupt handlers can be split into a linear first-level handler and a more complex second-level handler. This would justify the alternative policy to only accept drivers which pass the execution time analysis at compilation time. But we do not want to be that restrictive.

Thus our verifier can handle these cases as well. It can analyze the worst-case execution time of one loop iteration and insert run-time checks into the method's bytecode. The checks monitor the number of iterations and can terminate the method with an exception if the specified time limit is exceeded. Alternatively the run-time checks can be created by our bytecode to native code compiler. Then the verifier supplies the native code compiler with hints, where to add those checks. The advantage is, that hardware clocks can be used to measure the exact time used by the method, whereas the bytecode variant is based on estimated data.

4.4 Terminating drivers

When an interrupt handler is terminated, the corresponding driver is uninstalled. This is reasonable, because next time this interrupt occurs, the handler may exceed its time limit again. The termination of a driver leads to the following other problems:

- The data structures of a driver could be inconsistent.
- The device still generates interrupts, stressing the system.

To handle these problems a device driver must implement a *terminated* method. This method is called when the interrupt handler of the device is aborted. The task of this method is to clean up internal data structures and to reset the device.

The design of this method has to fulfill special requirements, because it is executed while interrupts are disabled. The verifier must be able to estimate the worst-case execution time of the method at compile time. If this is not possible, the driver is not allowed to be installed at all.

To fulfill these requirements the method can only contain loops where the number of iterations can be checked at compile time. In general, it should be possible to build a linear *terminate* method, so that this is not a serious restriction. If it is not possible, the *terminate* method can wake up another thread, which runs later with enabled interrupts and thus does not need to accomplish the strict requirements.

After cleanup, the system should be in an appropriate condition for installing another (or may be the same) driver for the device, so that applications which rely on the service of that specific device can still run.

5 Conclusion and future work

We can assure an upper bound execution time for every interrupt handler. This is done either at compile time or with checks at run time.

At compile time the verifier uses an approach of partial execution to determine the execution time. As a precondition a start value must be provided for each parameter. Up to now the result of our verifier is a table which contains the number of executions for each bytecode. To get a time information from this abstract data we need a mapping from the bytecodes to the worst-case execution time of each bytecode (cp. [PG01]). The simplest way to do this is a table of execution time information for each bytecode. This data depends on the target architecture and must be obtained in a calibration step.

If it is not possible to evaluate the worst-case execution time at compile time, we extend the critical methods with additional code. This code checks the elapsed time at run time and terminates the method if a specified amount of time is exceeded.

Thus our system is less vulnerable to erroneous interrupt handlers. In combination with the protection provided by the typesafe language Java, the occurrence of system crashes due to faulty device drivers is drastically reduced.

A side effect of the execution time limitation is that the system delay, due to the execution of interrupt handlers, is within certain limits. This may be profitable for real-time applications.

6 References

- | | |
|-------|--|
| Alt01 | M. Alt. Ein Bytecode-Verifier zur Verifikation von Betriebssystemkomponenten. Diplomarbeit (master thesis), University of Erlangen, Dept. of Comp. Science 4, July 2001. |
|-------|--|

- | | |
|--------|---|
| Bli94 | J. Blieberger. Discrete Loops And Worst Case Performance. <i>Computer Languages</i> , 20(3):193-212, 1994. |
| CY01 | A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In: <i>Symposium on Operating System Principles 01'</i> , 2001. |
| EE00 | J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In: <i>Proc. of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)</i> , Orlando, December 2000. |
| GFW+02 | M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX Operating System. In: <i>Proc. of the Usenix Annual Technical Conference</i> , Monterey, June 2002. |
| GKB01 | M. Golm, J. Kleinoeder, F. Bellosa. Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System. In: <i>Proc. of the 8th Workshop on Hot Topics in Operating Systems</i> , May 2001. |
| LY99 | T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Second Edition, Addison-Wesley, Reading/Massachusetts, 1999 |
| OS97 | G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In: <i>Proc. of SIG-PLAN 1997 Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)</i> , June 1997. |
| Par93 | C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. <i>Real-Time Systems</i> , 5(1):31-62, March 1993. |
| PG01 | P. Puschner, G. Bernat. WCET Analysis of Reusable Portable Code. In: <i>Proc. of the 13th International Euromicro Conference on Real-Time Systems</i> , June 2001. |
| PK89 | P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. <i>Journal of Real-Time Systems</i> , 1(2):159-176, September 1989. |
| Sha89 | A. Shaw. Reasoning about Time in Higher-Level Language Software. <i>IEEE Transactions on Software Engineering</i> , 15(7):875-889, July 1989. |

Extensible Distributed Operating System for Reliable Control Systems

Katsumi Maruyama, Kazuya Kodama, Soichiro Hidaka, Hiromichi Hashizume
National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
Email: {maruyama, kazuya, hidaka, has}@nii.ac.jp

Abstract

Since most control systems software is hardware-related, real-time-oriented and complex, adaptable OSs which help program productivity and maintainability improvement are in strong demand.

We are developing an adaptable and extensible OS based on micro-kernel and multi-server scheme: each server runs in a protected mode interacting only via messages, and could be added/extended/deleted easily. Since this OS is highly modularized, inter-process messaging overhead is a concern. Our implementation proved good efficiency and maintainability.

1. Introduction

Most systems, from large scale public telephone switching systems to home electronics, are controlled by software, and improvement of control program development productivity is necessary.

To facilitate the program development, suitable OSs are required. The followings are required to OSs for control systems.

- Required features largely depend on target fields. Extensible and adaptable OS is required.
- Control systems, such as telephone switching systems, require very severe multi and realtime processing capabilities. Very efficient multi-threading must be supported.
- Hardware controls are essential.

In most OSs, driver programs run in kernel mode, and their program development is very difficult.

- Control programs should be maintained for a very long time, adding new functions. Maintainability is of great importance.

General purpose OSs are not sufficient for these requirements. Therefore, in large and severe systems, specially designed OSs are used. In economical embedded systems,

small monitor-like OSs are used. However, these monitor-like OSs lack program protection mechanisms, and program development is difficult.

Therefore, an extensible/adaptable OS for control systems is required. We are developing a new OS characterized by:

- Use of an efficient and flexible micro-kernel (L4-ka).
- Multi-server based modular OS. (Each OS service is implemented as individual user-level process.)
- Robustness. Only the micro-kernel runs in kernel mode and in kernel space. Other modules run in a protected user space and mode.
- Hardware driver programs in user-level process.
- Flexible distributed processing by global message passing.

This OS structure proved to enhance OS modularity and ease of programming. However, inter-process messaging overhead should be considered. We measured the overhead, and the overhead was proved to be small enough.

2. Structure of this OS

2.1. The outline of this OS

Fig. 1 shows the structure of this OS. Thick square boxes represent independent logical spaces.

Micro kernel manages logical spaces (processes), multi threads, message passing (IPC) and interrupts. Only the micro kernel is executed in the kernel mode.

OS services, such as process management, file service, network service, are implemented by user-level processes, not by the kernel. They have their own logical space and are executed in user mode. Even hardware drivers are located in user-level processes. Processes interact only via messages.

2.2. L4 Micro kernel

We adopted L4-ka micro kernel [2] implemented at University of Karlsruhe, because it is very efficient and flexible. L4-ka is characterized by:

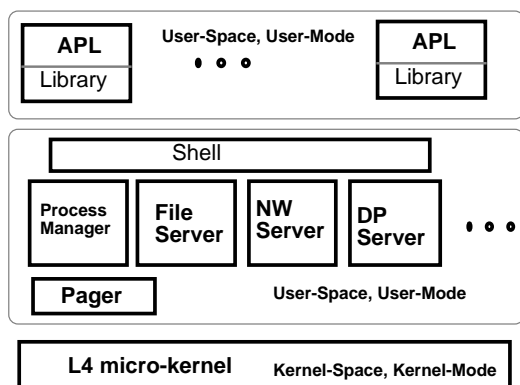


Figure 1. Multi server OS.

1. Efficient thread facilities.
2. Efficient and flexible message passing facilities: messages are sent synchronously to the destined threads by either value copy, buffer copy or page mapping.
3. Flexible memory management.

2.3. Logical space and pager

Pager is a user-level process to assign page frames when pagefault occurs. By rewriting the pager, a user can provide his own mapping algorithm. When a pagefault occurs, the kernel sends pagefault message to the pager. Pager assigns appropriate page frame, and reply message lets the kernel map new pages.

2.4. Service servers

Process manager manages allocation/freeing of logical space and processes. **File server** and **Network server** are implemented by rewriting those of Minix-OS [5].

2.5. Driver program

Fig. 2 shows the driver program structure. Each device driver has a **driver thread** which waits for request messages, and an **IRQ thread** which waits for interrupt messages. Drivers are executed in user mode. This helps facilitating their development. When the L4 kernel notices hardware interrupt, it sends an interrupt message to the driver, and driver action is activated.

In our implementation, HDD driver and ETH driver are included in File server and INET server, respectively. They can be separate processes, because drivers interact only by messages.

3. IPC overhead and file server

In this OS structure, IPC efficiency determines the system efficiency. IPC overhead lies in system calls and block

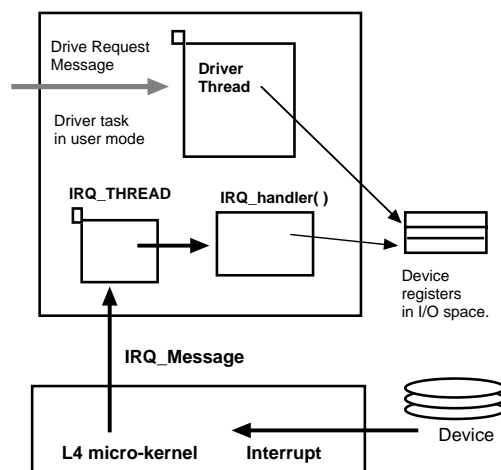


Figure 2. Driver program.

data transfers.

Prof. Liedtke et al. reported [1] a simple L4 message (inter address space, round trip) costs about 500 machine cycles, whereas the shortest Linux system call `getpid()` costs about 220 machine cycles, and that overall overhead will be several percent in comparison with monolithic OSs.

To evaluate the block data transfer overhead, we measured it in the case of file service. In monolithic OSs, file subsystem is included in the OS kernel; it accesses APL space directly and transfers data from/to APL by an efficient string copy operation.

In our OS, the **file server** is a user-level process, and it cannot access APL space directly. Data must be transferred using IPC, which may result in performance degradation. We measured the overhead and proved it small enough.

3.1. Buffer cache in file server

File server adopts **buffer caches** to improve file access speed (Fig. 3). Data blocks in HDD are identified by device number and block number. Recently accessed data blocks are cached in buffer caches, size of which is 1KB each. Buffer caches are located using hash table whose keys are device and block numbers. Data is transferred between an APL buffer and one or more buffer cache entries in FS.

3.2. IPC in L4 micro kernel

To transfer large volume data between processes, L4 provides an efficient buffer copy mechanism using temporal page mapping, and the following schemes can be used.

1. Straight transfer

Fig. 4 shows the program to transfer data block "a1" of process-A to buffer "b1" of process-B. The sender

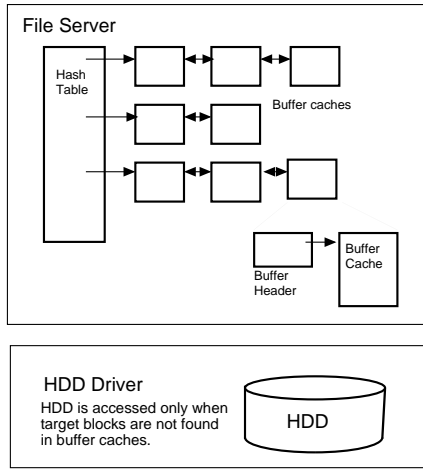


Figure 3. UNIX file subprogram.

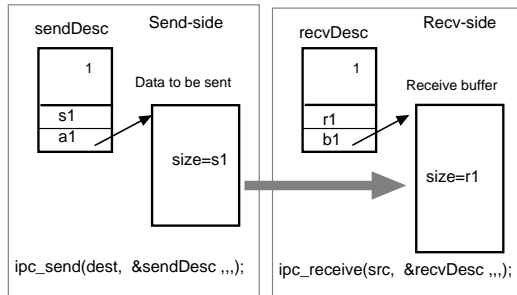


Figure 4. Straight buffer transfer.

prepares the send descriptor “sendDesc” and invokes `ipc_send()`. The receiver prepares the receive descriptor “rcvDesc” and invokes `ipc_receive()`.

2. Scatter/gather transfer

Sender and receiver can specify multiple buffers to send/receive data. Data is transferred in one IPC, automatically scattering and gathering data according to the send/rcv descriptors. In Fig. 5, data blocks “a1”, “a2” and “a3” of process-A are transferred to the buffer “b1” of process-B.

3.3. File server and data transfer

Let’s assume that APL is requesting to read block data of 2.5KB. As each buffer cache is 1KB in size, requested data is cached in 3 buffer caches. Therefore 3 IPCs are required in a straight transfer.

In the scatter/gather transfer, data in multiple buffer caches can be transferred in one IPC. Fig. 6 shows this scheme.

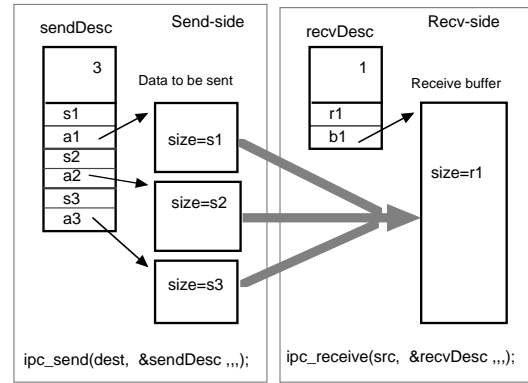


Figure 5. Scatter/gather buffer transfer.

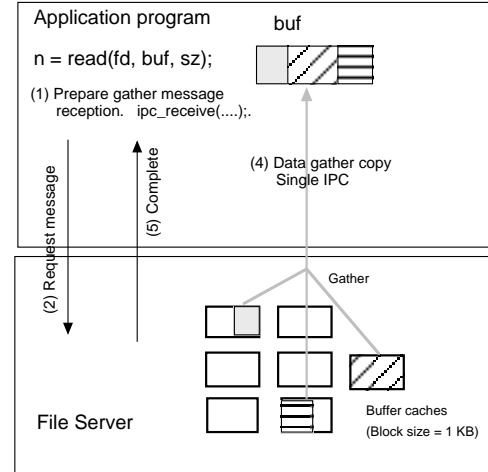


Figure 6. File read using scatter/gather transfer.

3.4. Evaluation

To evaluate the file-read IPC overhead, the following test program is used:

```
for (i = 0; i < 100000; i++) {
    lseek(fd, 0, 0);
    read(fd, buf, size);
}
```

This measures the time to transfer data from file server buffer caches to APL space (CPU=800MHz Pentium-3, L2 cache size= 512KB). At the first access, all data are copied on buffer caches from HDD, so that HDD access time are amortized.

Fig. 7 shows the results. The same program is tested on Minix-OS (Version 2.0) and Linux (Kernel version 2.2.12), and results are shown as Minix(3) and Linux(5), respectively.

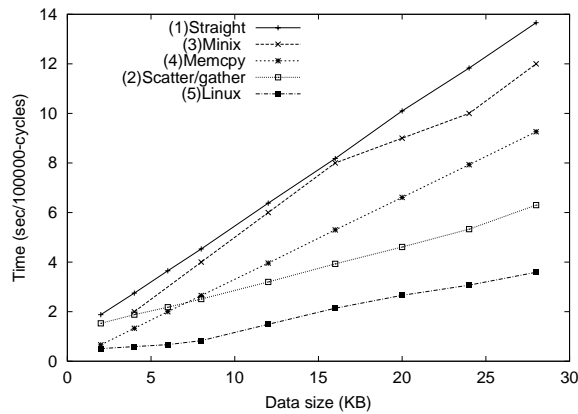


Figure 7. Inter-process communication overhead.

In Minix, file server delegates copy to system task who has direct access to resident physical address space of APLs and servers, so no IPC is used for data transfer per se. On the other hand, Linux is a highly optimized monolithic OS.

Memcpy(4) shows the string copy using ANSI memcpy function (in the same logical space) for reference.

This test shows that:

- Straight transfer and Minix are comparable in transfer speed.
- Scatter/gather transfer improves speed significantly.
- Inter-process scatter/gather transfer is faster than intra-process memcpy(). This is because the former copies data word by word using temporary page mapping, and the latter copies data byte by byte.
- Linux is highly optimized and very fast.

Therefore, block data transfer overhead in concern was small enough in the scatter/gather transfer scheme.

4. Distributed processing

In this OS, all APLs and OS servers interact only via messages. We are now designing a distributed processing server, which delivers messages globally according to global thread IDs. This server would enable file server to reside on remote hosts without redesigning it. Fig. 8 compares this OS and NFS architecture.

In control systems, remote resource control is important. In this way, remote resources are easily controlled.

5. Conclusion

We are implementing this OS on IBM-PC. Through this OS implementation, the followings are confirmed:

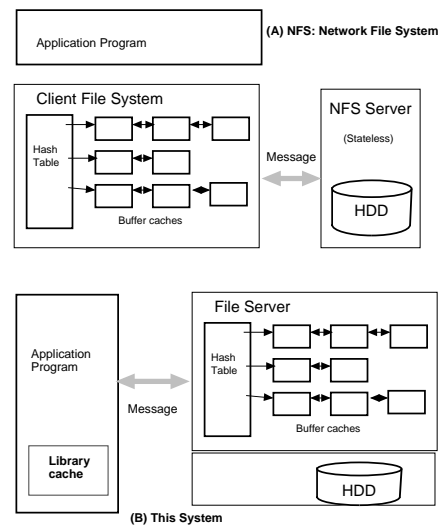


Figure 8. Distributed File Server.

Modularity and Extensibility New functions, which require kernel modification in monolithic OSs, can be easily extended only by adding user-level process in independent address space.

Robustness Most functions are implemented by protected user-level processes.

Ease of program development OS services, even a hardware driver, can be implemented by user-level process. Compared to kernel programs, user-level programs are easy to develop.

Hardware control Hardware control drivers are also user-level, and their development is facilitated.

Distributed processing We are now trying to extend the message passing to global scope. With global messaging, resources located in remote hosts can be accessed as local resources. This distributed processing is useful in control systems.

Acknowledgments We would like to thank Dr. Akira Nakamura at the International Christian University, Dr. Yusheng Ji, Dr. Ichiro Ide for their valuable discussions and suggestions.

References

- [1] Jochen Liedtke. Improving IPC by kernel design. In *Proc. of SOSP'93*
- [2] Jochen Liedtke. On μ -Kernel Construction. *Operating Systems Review*, Vol. 29, No. 5, pp. 237–250, December 1995.
- [3] IBM Watson Research Center. SawMill home page <http://www.research.ibm.com/sawmill/>
- [4] Hermann Härtig, et al. The performance of μ -Kernel-based systems. In *SOSP97*.
- [5] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice-Hall

Fault Tolerance and Avoidance in Biomedical Systems

Shane Stephens & Gernot Heiser
School of Computer Science and Engineering
University of New South Wales

Abstract

It is important for a variety of reasons that biomedical systems execute without errors. One useful approach towards error-free software is to design a range of fault tolerant properties into applications software. In addition, by restricting the behaviour of an application and requiring explicit allocation of resources such as memory, errors can be caught while an application is still being written, rather than once an application has been released. This paper investigates how an operating system can support biomedical applications using these approaches.

1 Introduction

A biomedical system is one which interfaces with humans in a medical context. Examples include life-support devices such as pace makers, diagnostic and monitoring devices such as electrocardiographs, and prosthetic limbs and organs.

Fault tolerance in this context refers to recovery from a fault in such a way that the faulty service can recover, and continues to be offered to the user. Fault avoidance refers to increasing the stringency of a system in such a way that bugs are easier to find.

While fault tolerance and avoidance is important in all systems, it is especially important to ensure that biomedical devices work as intended. In some cases, the patient is unable to survive without the assistance of the device, while in others, the patient relies upon the correctness of information the device provides. Therefore the methodologies used in conventional software are inappropriate for Biomedical software.

One approach to fault tolerance is formal verification of code [3]. The author is currently involved in an attempt to verify the L4 microkernel [4], and we are confident that our approach will succeed. Another feature of L4 that sets it apart from traditional embedded kernels is that it implements memory protection. Memory protection is useful in this context as it limits the damage that can occur due to malfunctioning code.

However, executing on a verified kernel is not sufficient protection for biomedical systems - even in the presence of a perfect kernel, user applications can fail. One solution to this problem is to require that the user applications themselves be verified in a similar manner to the kernel. Given the magnitude of effort required to verify code, however, this may not be a practical approach: there are many more user applications than kernels.

Fault tolerance and avoidance should therefore be examined as an alternative in cases where full verification of user code is not a feasible option. It is a central thesis of this paper that an appropriately designed operating system can support and aid programmers of user applications who wish to write fault tolerant code.

A prototype version of such an operating system, Biomedical Operating System (BiOS), has been written, and further research is currently in progress. Some key aspects of this operating system are presented below.

2 The BIOS Design

BiOS provides a small set of user-level services, implemented on top of L4. These services include a pager, a system-call server, and a packet server. Given the size and modularity of these services, verification should be possible.

The domain of embedded biomedical applications has several important properties that have influenced the design of BiOS. These properties are:

- that embedded biomedical applications typically require only a small number of concurrent threads of execution. A general purpose system that allows arbitrary execution of multiple applications is not required.
- that biomedical devices will typically not be required to run code which was not produced by the developer of the device. In general, faults will occur because of programming bugs, not malicious code.

- that typical biomedical applications involve the continuous or semi-continuous processing of packetised data. This is exploited by the provision of a highly optimised streams abstraction which lies at the base of many of the fault-tolerant properties of BiOS.
- that because of limitations in human perception and the relatively slow rate of events within the human body, most biomedical applications have a data acquisition/production frequency in the order of only tens of Hertz. In addition, the human body itself adapts gracefully to delayed deadlines on the milliseconds scale - jerky video streams are still watchable, and a delay between action and effect can often be adapted to. Hence hard real-time guarantees are not required in general.

Given these properties, a decision was made to base BiOS inter-process communication around a streams abstraction. Although this abstraction is quite different to existing UNIX abstractions, BiOS is not intended to be a general-purpose operating system. In addition, provision of this abstraction allows biomedical developers to think about streams-based problems in a more natural manner. Finally, soft real-time schedulers for streams exist (see for instance Löser et. al. [5]), and adaptation of an existing scheduler to the BiOS system should be possible if soft real-time is required.

BiOS streams connect several participating threads together in an ordered fashion. When a stream is created, it is initialised with a fixed number of packets that can be passed along the stream. At any given time, each packet may only belong to one thread (or “stream element”). Adjacent stream elements communicate by transferring ownership of a packet from one element to another.

This promotes a user view of an application as a set of communicating, modular stream elements. Ideally, each element performs a single logical action, and each logical action is distinct in its execution from the rest of the system.

BiOS enforces this abstraction by providing only a streams interface to the system drivers. This also increases the efficiency of the system - BiOS streams are designed to provide a zero-copy communications mechanism.

3 Modularity

Providing applications developers with a system interface that promotes modularity has several advantages. Firstly, the task of writing an application is simplified, as the design approach essentially consists of identifying candidate stream elements, designing an interface between adjacent elements, and writing each element.

Secondly, modular applications are easier to debug, as accidental memory accesses are more likely to cause a protection fault than a side-effect.

Finally, modular applications are easier to verify [2] (if verification is considered absolutely necessary), as the application is already split into a set of orthogonal sections that communicate via a well-defined interface.

4 Protection vs Performance

To provide an efficient zero-copy mechanism for streams, BiOS must place all packets in globally shared memory. However, this weakens interprocess protection, because stream elements can write to packets that they do not own.

To solve this problem without sacrificing performance, BiOS provides two completely separate implementations of the streams interface. The first implementation enforces protection by manipulation of virtual memory using an L4 memory primitive known as “grant”. Grant operates on one or more contiguous pages of memory, and can be thought of as a transfer of the underlying frames from one address space to another.

The safe streams interface provides an operating system service known as the “packet server”. When a new stream is created, that stream’s packets are initialised within the packet server’s address space, one per page, and are only granted to participating threads as required. Similarly, when a thread decides to send a packet, this packet is granted back to the packet server. In this manner, illegal accesses within the region of memory containing the packets are detected by the system, and the developer is notified.

However, due to the relatively high cost of page granting, this implementation is quite slow. Given the nature of many biomedical applications, this limitation may not be significant. However, if more efficient communication is required, BiOS provides a second implementation of the streams interface. This implementation provides a permanently mapped region of memory for the stream. Packets reside in this region, and illegal accesses are not caught.

This interface is fast for three reasons. Firstly, expensive virtual memory operations are not required. Secondly, because the user applications are given a pointer to a buffer rather than supplying one, the stream can be used to implement zero-copy transfer of data all the way along the stream (including to and from operating system drivers). Finally, the operating system does not play a heavy role in the streams mechanism (being involved only in blocking stream elements that are waiting on packets which have not been sent), which reduces execution time substantially.

Because the two implementations provide exactly the same interface, switching from the safe implementation to the fast implementation simply requires toggling an initialisation flag. As a result, user code which executes safely on the slow interface can still be considered safe when running on the fast interface.

It is evident that carefully written malicious code could seem to execute correctly on the protected implementation, yet perform illegal accesses on the high-performance implementation. However, the purpose of the BiOS dual streams implementation is not to protect against malicious code, but instead to detect accidentally programmed bugs. This restriction explicitly excludes consideration of Byzantine failures.

5 Fault Recovery

The programmer's view of BiOS applications is that of a cooperating system of stream elements. This view allows the programmer to implement several fault recovery mechanisms at user level with a minimum of difficulty.

BiOS can be configured to restart a task when an exception is raised by that task. Rather than using the 'main' entry point, BiOS will start the task at an additional, user-defined entry point (much like a light-weight version of UNIX signals). All stream memory and mappings in the task are preserved, and the user code must then determine what error occurred and handle the error appropriately.

A simple mechanism for dealing with an error may be simply to discard the most recent packet of data and request the next one. Alternatively, the user module may simply be restarted with all of its state re-initialised. More complicated mechanisms may simply attempt to process the faulty packet with an alternative algorithm, or execute an internal consistency check before continuing.

The user can also insert stream elements which perform explicit bounds-checking at various points of the stream. These elements can be registered with BiOS as additional exception-generators, and can be programmed to trigger if packets are detected with erroneous or nonsensical data.

Such stream elements could look for signs of malfunction such as packets that contain unexpected values (for instance, negative values in a frequency field); or an unreasonable time without a new packet becoming available. Other user-defined signs could also be implemented if required.

This approach provides mechanisms by which users can write fault-tolerant code, rather than dictating operating-system level fault-tolerant procedures to the programmer.

6 N-Version programming

N-Version programming is a popular existing technique for writing fault-tolerant software where proof of an algorithm is impractical. Essentially, the approach consists of processing data with several implementations of the same algorithm, and attempting to find a consensus of the results [1].

This approach can readily be implemented with little overhead using BiOS streams. Several alternate implemen-

tations of the required algorithm can be implemented as separate threads or processes, and registered in a stream. A demultiplexing stream element can then make several copies of a packet and pass a copy to each implementation. Finally, a consensus element could collect each implementation's result, and use any of the existing approaches to choose an acceptable outcome based upon the results gathered.

7 Conclusion

Provision of a reliable system is the responsibility of both the operating system provider and the application writer. This paper has examined some operating system features that can aid the application writer in construction of a fault-tolerant application.

References

- [1] A. Avizienis. The methodology of n-version programming. In *Software Fault Tolerance*, pages 23–46, 1995.
- [2] K. Havelund and J. Skakkebaek. Practical application of model checking in software verification. In *Proceedings of the 7th Workshop on the SPIN Verification System*, Sept. 1999.
- [3] C. A. R. Hoare. An axiomatic approach to computer programming. *Commun. ACM*, 12:576–580, 1969.
- [4] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-02-März 2002, Dresden University of Technology, 2002. Available from URL: <http://os.inf.tu-dresden.de/vfiasco/>.
- [5] J. Löser, H. Härtig, and L. Reuther. A streaming interface for real-time interprocess communication. Technical Report TUD-FI01-09–August 2001, Dresden University of Technology, 2001.

Gaining and Maintaining Confidence in Operating Systems Security

Trent Jaeger Antony Edwards Xiaolan Zhang
IBM T. J. Watson Research Center
 19 Skyline Drive, Hawthorne, NY 10532 USA
 Contact Email: {jaeger}@us.ibm.com

March 15, 2002

1 Introduction

Recently, there has been a lot of work in the verification of security properties in programs. Engler et al. use static analysis to find flaws in the implementation of Linux device drivers, such as the failure to release locks [4]. Edwards et al. use static and dynamic analysis to verify that the authorization hooks of the Linux Security Modules (LSM) framework are placed such that all the necessary authorizations are performed [2, 12]. In addition, Shankar et al. and Larochelle et al. show how to use static analysis tools to find program vulnerabilities, such as buffer overflows and printf vulnerabilities [7, 10, 11]. Lastly, Necula et al. show that we use detect and leverage the cases in which C is used in a type-safe manner in order to detect memory errors [9]. Runtime verification can be used to detect errors in other cases.

While these tools demonstrate that we can use automated tools to gain some level of assurance for our programs, these tools are aimed at individual errors (e.g., static analysis buffer overflows), require significant skill to use properly (e.g., require ad hoc analysis code to find particular driver errors or to write exploits that identify real problems), and are limited to finding only some types of errors in their class (e.g., static analysis is limited by the type safety of C). Ultimately, the goal is to gain a satisfactory level of assurance of the security of the entire program and maintain that level of assurance over the program's evolution (ideally, to improve its level of assurance). Given the breadth and depth of verification approaches, we believe that it is time to examine how they can be put together into a coherent approach for program security assurance.

We contrast the approach of performing assurance by a concerted application of verification tools with the traditional approach to establishing operating system secu-

rity assurance. The Orange Book and its successor, the Common Criteria, achieve assurance by requiring detailed documentation and design for security from the onset of the project [8, 6]. This security focus is to then guide the transformation of these specifications into an implementation. Some testing and code review is done, but these are ad hoc processes. Further, significant modifications to an assured system are not permitted because they would demand a new labor-intensive assurance, and this approach cannot be applied to already-constructed systems. Not surprisingly given the effort and cost of this form of assurance, very few systems have been built to the higher levels of assurance.

In this paper, we examine some of these verification approaches and develop an approach to operating system security assurance based on automated verification tools. The aim of the approach to enable: (1) practical verification of key security properties; (2) extension in both the types and number of verification tools to improve the breadth and quality of verification over time; and (3) management of verification state to enable maintenance of assurance as the system evolves. We first examine some verification problems in Section 2 to identify the types of tasks that need to be performed in verification. We then examine how individual verification tasks are performed now in Section 3. Then, we propose an approach for performing the necessary analyses and discuss how current analyses may be extended to make this approach work in Section 4.

2 Security Verification Problem

The history of program correctness verification does not generally have a good reputation, but a significant set of program analysis have been constructed that yield practical results. In general, these tools typically perform focused analysis for particular program properties (e.g., Y2K bugs [3], null pointers [5], etc.).

As a result of this experience, we do not believe that security verification is a single process, but rather, a series of analyses that yield greater confidence in the security offered by a program. Also, as new vulnerabilities are discovered, new analyses must be added to the verification toolset in order to maintain confidence in the program. Further, such verification tools should enable practical regression testing of security properties as the program evolves.

For verification that the Linux kernel enforces a system access control policy correctly (i.e., is a correct reference monitor), we identify the following criteria:

- **Verify code integrity:** Detect vulnerabilities that would enable an attacker to run their attack code instead of program code (e.g., buffer overflows, printf vulnerabilities, etc.).
- **Verify complete authorization:** Verify that all controlled operations are executed only after all the necessary authorization requirements are checked. Note that whether permissions to perform these operations are granted to a principal is a policy question, so policy verification is a separate, necessary task that we do not discuss further here.
- **Verify access using authorized object only:** Detect vulnerabilities where an attacker can switch the object used in controlled operations between the authorization and use (i.e., time-of-check-to-time-of-use or TOCTTOU attacks [1]).
- **Detect permission leakage:** Verify enforcement of system invariants necessary to prevent permission leakage, such as closing the necessary descriptors on an exec.

If these four criteria are verified effectively, then only the operations authorized by the system's access control policy can be performed. The kernel integrity is protected, all operations are correctly authorized, all accesses use authorized objects, and there are no errors that leak permissions. Thus, relative to the confidence we have in the verification of these criteria, the Linux kernel is assured to enforce the system security policy.

The problem then is to define and implement the analyses that verify these criteria with a reasonable degree of confidence. While perfect static verification of a kernel is impossible given that it is written in C and assembler, a variety of approaches can be employed in unison to improve confidence in the assurance. For example, we use static analysis to detect possible TOCTTOU vulnerabilities [1] by identifying any variable used in a controlled operation that has not been authorized since it

was last assigned. This analysis identifies variables as potential vulnerabilities even if they are extracted from an authorized object. While this analysis is conservative in a type safe language, it is possible to modify the value of the variable through non-type safe programming. At present, we assume that the kernel developers are trusted, but ultimately, identification of non-type safe code and verification that this code does not access the stack (for local variables) is necessary to assure this property fully. Necula et al.'s work can find where type safety is not preserved, so other analyses can be built to better verify that such an attack is not possible or only possible in places where runtime checks can be inserted.

Further, a conservative analysis means that several false positives may be identified. Currently, these are examined manually, but many of these are really the same situation, such as extraction of inodes from authorized dendries. Some secondary analyses can make such verification practical. Lastly, since such analyses can be rerun as the kernel evolves, the assurance of the kernel is maintained even as the kernel changes.

3 Verification Experiences

In this section, we outline two example analyses to demonstrate the types of verification tasks that can be done and their effectiveness¹.

3.1 Linux Security Module Verification

The goal of Linux Security Modules (LSM) verification is to ensure that any security-sensitive operation in the Linux kernel is authorized for its proper requirements via the LSM authorization hooks. Since access to such operations involves access to high-level kernel data objects (e.g., inodes, sockets, etc.), we identify any access to these data structures as a mediation point to be authorized (i.e., a *controlled operation*).

We use a runtime analysis tool to identify authorization requirements for the controlled operations and anomalies to those requirements []. To start, we assume that LSM is largely correct, so we can express invariants that indicate an anomaly in the authorization. An example of an anomaly is a controlled operation that has different authorizations for different runs of the same system call. The other controlled operations are classified as consistent, but since we may be missing an authorization entirely there can still be an error. Since all controlled operations usually require the same authorizations, these

¹We would expect to expand this section for the full paper, so more experiences can be analyzed.

errors are easy to identify. For each anomaly, we must determine if there is an exploitable situation. At present, we have found 5 significant anomalies, where the LSM community agreed that 4 are truly errors and the other case works under the limited circumstances intended.

Runtime analysis is limited by the statement coverage and input value coverage provided by the benchmarks. In the first case, we have found that performance benchmarks only execute 20% of the Linux kernel statements. Further, the ability to leverage a possible TOCTTOU situation requires active attacks to provide the inputs necessary for the analysis to see the changed value. Therefore, we also use static analysis techniques to find cases where controlled operations are performed using variables that are not authorized as expected. Most of these cases turn out to be possible TOCTTOU vulnerabilities where a variable is authorized, but it is re-computed from higher-level objects rather than used directly. Again, exploits must be written to determine whether such situations are vulnerabilities or not.

When the kernel is modified, we can verify that the authorization requirements and no TOCTTOU vulnerabilities have been added. Modifications to files and functions indicate the scope of regression testing. In general, all the system calls that have code paths that intersect the modified code may need to be tested, although some optimization are possible for static analysis. For example, when no new controlled operation variables or code paths are introduced and we can see that the order between the authorization and the operation is maintained then verification is not necessary.

3.2 Buffer Overflow Detection

Wagner, et al. have developed a static analysis tool to detect potential buffer overflow vulnerabilities in C code [11].

In their approach, C strings (character arrays) are modeled as an abstract data type manipulated via the standard C library functions (e.g. `strcpy`, `strcat`, `sprintf`). Therefore, buffer overflows caused by manipulating strings directly cannot be detected, however, they claim that this represents only a small portion of the vulnerabilities.

Each string in the program is associated with two ranges. One range stores the number of bytes allocated to the string, the other stores the number of bytes in use. C string functions are modeled by their effect on these ranges. Each integer variable in the program is also associated with a range of possible values. The tool then performs a, flow-insensitive, integer range analysis that

maintains these ranges, and checks for violations of the safety property: $\text{in_use}(s) \leq \text{alloc}(s)$.

Flow-insensitivity was chosen to allow efficient analysis of large programs. Unfortunately, it also leads to a large number of false-positives that must be manually inspected by a human. Several vulnerabilities were found, and they also identified some that were found not to be exploitable.

4 Verification Approach

Assurance consists of running analyses to verify the four types of security properties. The first and fourth security properties are ad hoc, so multiple analyses may be performed for each. For example, buffer overflow and printf vulnerabilities involve different analyses.

In general, each analysis consists of the following:

- **Scope:** Each analysis works under a possibly null set of assumptions (e.g., type safety). Obviously, the fewer assumptions an analysis depends on, the broader its scope.
- **Scope verification:** Therefore, other analyses may be necessary to maximize the likelihood that all assumptions hold.
- **Classifications:** Each analysis classifies the relevant cases (e.g., positive and negative). Most analyses have some false positives, but a good analysis will have a manageable set of false positives and no false negatives.
- **Classification analysis:** Subsequent analyses may be necessary to verify that a classification is correct with respect to the security requirements (e.g., deriving and running potential exploit programs for positive cases).
- **Case dependencies:** Each analysis result depends on some conditions that, if unchanged, do not require the regression testing of a case upon system modification.

First, each analysis may depend on certain assumptions. For LSM verification, we assume that all controlled operations are performed on variables of controlled data types. Thus, the analysis can handle a variety of bizarre type castings, but cannot detect accesses through other data types, such as `char *`. Initially, we assume that kernel developers are trusted not to do such things, but we would ultimately like to leverage Necula et al.'s approach to protect against non-type safe code [9]. These

analyses will likely have some probabilistic nature and be highly domain-dependent. For example, for a particular function that is not type-safe, we may want to detect whether it can ever access particular controlled data objects.

The main goal of each analysis is to classify its cases into positive (i.e., likely errors) and negative (i.e., likely correct) cases. For all but the simplest analyses, false classifications are possible. In LSM verification, we are conservative about our static identification of positives and do not generate any false negatives (under our analysis scope). This is not the case with the buffer overflow detection where a small number of false negatives are permitted. Then, we have to perform subsequent analyses on positives. Currently, such analyses consist of manual inspection and exploit generation in both tools. For LSM verification, some manual inspections, such as initialization functions and extraction of inodes from checked dentries, should be automated easily. Further, we envision moving to templates for exploit generation, similar to using templates to describe attacks [1]. We would identify system calls that enable modification of relationships (e.g., descriptor to file via `dup`) and means for triggering reschedules (e.g., forcing page faults) to implement TOCTTOU attacks.

Thus, the analysis results in classifications into positives, negatives, and false cases of each. As the kernel is modified, we would like to enable system regression testing commensurate with the extent of the changes. Minimizing the effort involves eliminating the cases where the factors that determined its classification are not changed. There are two sets of factors. First, all classifications depend on the execution context in which the case is run. For LSM verification, each controlled operation is run in a system call path and is authorized by certain LSM hooks, so as long as these remain fixed re-verification is not necessary. Second, the reasons that cases are falsely classified determine whether they will be again. For LSM verification, initialization cases can be easily identified, and we can verify that an association between dentries and inodes is still permanent (e.g., because the inode field in the dentry is never reset).

5 Conclusions

We develop an approach for operating system assurance, in particular the Linux kernel's ability to serve as a correct reference monitor, extrapolated from current research in security property verification. We find such analyses currently enable identification of vulnerabilities, but work under a variety of assumptions and require significant effort to use. We propose an approach

whereby assumptions are justified, a sequence of analyses enable complete classification of cases, and regression testing is possible. The approach enables further development of analyses into a coherent framework, so the system's assurance can be determined with confidence, this confidence can be enhanced with improved analyses, and this confidence can be maintained when the system evolves.

References

- [1] M. Bishop and M. Dilger. Checking for race conditions in file accesses. Technical Report CSE-95-10, University of California at Davis, September 1995.
- [2] A. Edwards, T. Jaeger, and X. Zhang. Verifying authorization hook placement for the Linux Security Modules framework. TR 22254, IBM, December 2001.
- [3] M. Elsmann, J. S. Foster, and A. Aiken. Carillon – a system to find Y2K problems in C programs, user manual. www.cs.berkeley.edu/carillon, 1999.
- [4] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operation System Design and Implementation (OSDI)*, October 2000.
- [5] D. Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1996.
- [6] ITSEC. *Common Criteria for Information Security Technology Evaluation*. ITSEC, 1998. Available at www.commoncriteria.org.
- [7] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Tenth USENIX Security Symposium*, 2001.
- [8] NCSC. *Trusted Computer Security Evaluation Criteria*. National Computer Security Center, 1985. DoD 5200.28-STD, also known as the Orange Book.
- [9] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the Principles of Programming Languages*, 2002.
- [10] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the Tenth USENIX Security Symposium*, 2001.
- [11] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS Network and Distributed System Security Symposium*, 2000.
- [12] X. Zhang, A. Edwards, and T. Jaeger. Using CQual for static analysis of authorization hook placement, February 2002. Submitted for conference publication.

High-Confidence Operating Systems

Radu Grosu, Erez Zadok¹, Scott A. Smolka, Rance Cleaveland, and Yanhong A. Liu
Stony Brook University

1 Introduction

Operating systems (OSs) are among the most sophisticated software systems in widespread use, and among the most expensive and time-consuming to develop and maintain. OS software must also be robust and dependable, since OS failures can result in system crashes that corrupt user data, endanger human lives (cf. embedded systems), or provide open avenues of attack for hackers or even cyber-terrorists.

OSs present their designers with enormous development challenges. On the one hand, many activities inside an OS happen concurrently: caches are flushed periodically; processes and threads are stopped and restarted; interrupts and other signals arrive at random times and must be handled promptly; data can be transferred through multiple channels (memory, DMA, I/O buses). Concurrent processing introduces well-known difficulties into the traditional code-test-debug paradigm, since errors can be difficult to repeat owing to race conditions between concurrent processes. On the other hand, debugging even sequential OS modules poses difficulties, since the OS's closeness to the actual computing hardware requires *in situ* testing.

It is also important that OS *system-call interfaces* be well documented so that they may serve as useful design guides for OS implementers and as interface definitions for application developers. Man pages, currently the primary source of documentation for system-call interfaces, are often incomplete, vague, ambiguous, and even incorrect. Another often under-appreciated aspect of OS software is the profusion of different OSs in use, particularly in the embedded-systems arena where OSs such as Vx-Works or pSOS or (more often) proprietary ad hoc OSs are deployed.

The above observations highlight the great impact that improved OS development techniques would have on all enterprises that produce or use OS software. If man pages could be guaranteed to be correct and complete; if system calls could be certified to be free from deadlocks and memory leaks; if causes of system crashes could be quickly diagnosed; then the savings to the OS and application-development communities would be enormous. If this could also be accomplished while reducing OS development costs, then the impact is even greater. We refer to this ideal—better OSs at lower cost— as *High-Confidence Operating Systems* (HCOS).

In this paper, we present an overview of our work on bringing the HCOS concept to bear on the practice of OS development. Section 2 presents the overall methodology we are pursuing, a central component of which is the Concurrent Class Machines (CCM) modeling formalism. Section 3 describes how we are using CCMS to model system-call man pages. Section 4 discusses how we verify our CCM models against different kinds of requirements. Section 5 concludes with a status report.

In related work, efforts to validate OSs fall into three main categories: verification techniques [3], compilation techniques [4, 7], and external runtime testing [6]. The references given are a sampling. The HCOS approach focuses on the formal modeling of OS system calls and their interfaces, and utilizes newly developed techniques from all three categories.

2 Methodology

The organizing principle of our approach is that *an ounce of modeling is worth a pound of debugging*. In particular, we advocate the use of *formal operational modeling* as a methodology that can fundamentally and dramatically improve how OS software, and indeed any low-level system software, is developed. We envision these models

¹Contact author: Erez Zadok; Computer Science Department, 1416 Computer Science Building, Stony Brook University, Stony Brook, NY 11794-4400; Phone: +1 631 632 8461; Email: ezk@cs.sunysb.edu

being used throughout the OS development and deployment process, as active (i.e., executable) documentation for designers and application developers; as mechanically analyzable requirements and design artifacts; and as bases for reliable implementations.

The specific modeling formalism we are using, *concurrent class machines* (CCMs) [5], extends basic finite-state machines with features capturing a variety of object-oriented (OO) concepts, including classes and inheritance, objects and object creation, method invocation and exceptions, multithreading, guarded commands, and abstract collection types. The CCM model builds on our previous work in the formal modeling of hierarchic reactive systems, e.g. [1], and provides an intuitive, graphical notation for modeling system behavior at different levels of abstraction. In contrast with existing OO design notations, CCMs also possess a mathematically precise operational semantics that defines the execution steps that CCM models can engage in; this semantics makes CCM models candidates for a variety of different mechanical analyses. Figure 1 shows how CCMs provide a uniform basis for requirements analysis, verification, and code generation:

- **Executable and analyzable man pages:** CCMs model system-call interfaces and system properties, such as deadlock and livelock freedom. Unlike man pages, the resulting specifications are graphical, executable, precise and unambiguous.
- **Verifying models against requirements:** Verification techniques are used to check whether the CCMs derived (via compilation) from system-call implementations correctly implement man-page-derived system-call interface and requirements (required properties) that are also given as CCMs.
- **Models as system monitors:** Automatic code generation based on CCMs is used to produce efficient code for monitoring the runtime behavior of OS implementations in order to detect and, in some cases prevent, erroneous behavior.

A central idea in our approach is that of an *instrumented CCM*, where a CCM describing the implementation of an OS system call is combined with the CCM of the corresponding system-call interface or CCM of the requirements. Man-page CCMs can also be instrumented by combining them with requirements CCMs they must adhere to.

3 Operational Modeling of Man Pages

The process of determining the exact behavior of a system call begins with a careful reading of its documentation, including an inspection of the arguments that are passed to the system call, its return values, and their types. Man pages typically specify valid inputs and expected return values. The latter are divided into values that indicate success and values that indicate failures, or *exceptions*. Based on this information, an initial mock-up of a CCM can be developed.

Figure 2 shows an example, the modeling of the `creat` system call as a method of the `FileSystem` CCM. Let us first describe the visual notation. A class machine is a named rounded box that has several compartments: one for attributes and one for each method. A method has an entry point (hollow circle), several exit points (filled circles) and several exception points (filled diamonds). Exit and exception points may be marked with an expression denoting the return value. The entry point is connected to the exit (and exception) points by transitions and method invocations. A transition (shown as a labeled arrow) is an atomic guarded assignment. A method invocation (shown as a rounded box) has an entry point, an exit point and several exception points. Exit and exception points may be marked with variables to hold the return values. Exceptions propagate by default to the enclosing levels. A method may contain local variable declarations. As in UML, an attribute or method marked with + is public.

The `creat` method takes as arguments a pathname `pn` and a mode `m`, splits the first into a path `p` and a name `n`, creates a new file with path `p`, name `n` and mode `m` and returns its file descriptor `d`. If a file already exists at `pn`, `creat` truncates it to zero bytes. However, the call only works for regular files, not directories (for which the user should use `mkdir`). We reflect this condition in the CCM as follows: if the file name to be created already exists and the type of that file is `dir`, then exit this system call with the error condition `EISDIR`; otherwise continue to the next step in the CCM.

In cases where the man pages are insufficiently detailed or known to be inaccurate, we inspect the actual kernel sources at or near the entry point of that given system call into the kernel. For example, the manual page for the `creat` system call (on Red Hat Linux 7.1) does not specify that it will return an `ENOQUOTA` (quota exceeded) error code if the user's quota was exceeded when trying to add the new file; or that it will return `EIO` (I/O error) if a hardware failure occurred while trying to add the new file's entry to the on-disk directory. We found these conditions by inspecting the kernel sources for a

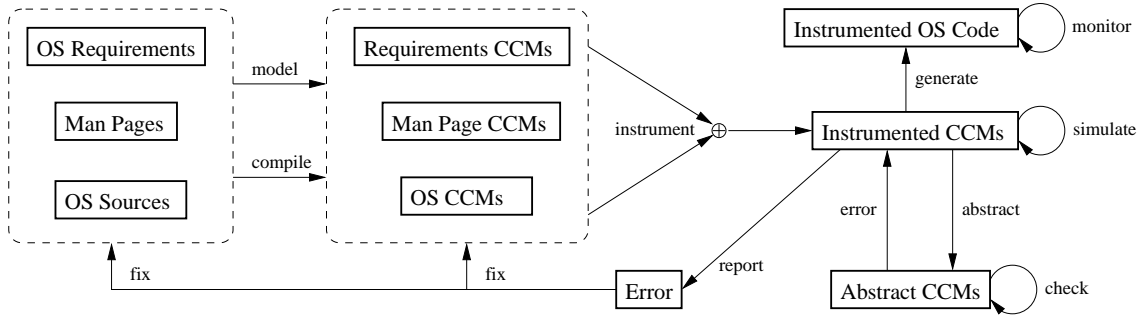


Figure 1: HCOS methodology overview.

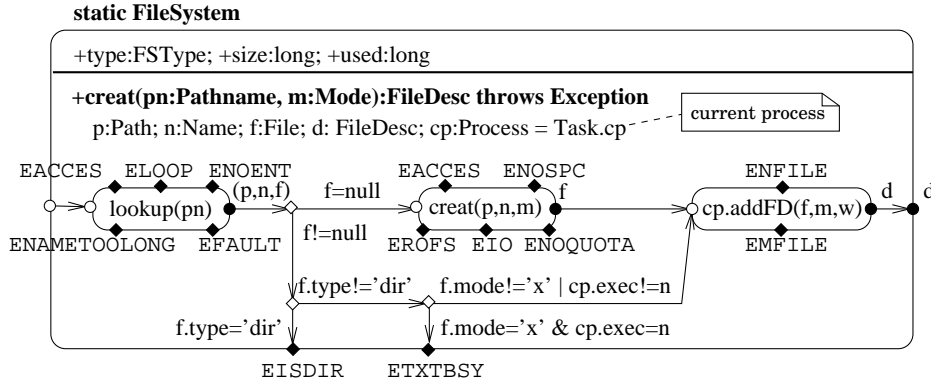


Figure 2: The CCM for the creat system call.

running version of Linux. Incorporating this behavior into the CCM above is straightforward, and the result is a more complete accounting of the behavior of `creat` than is available from its man pages. The description is also much more concise; instead of the several pages of text used to document `creat`, the diagram fits onto less than half a page.

CCM models have several benefits. In addition to their clarity and conciseness, they are *executable*, which enables application and OS developers to experiment with system calls on different inputs to see how they behave. CCMs also permit inter-system-call analyses in general, and those involving concurrency in particular, to be rigorously studied. For example, suppose one process tries to read a (shared-mapping) memory-mapped file, while another process tries to write to that same file asynchronously not using the `mmap`-interface. The OS must carefully coordinate the transfer of (possibly cached) data, lock pages and files, to ensure data coherency. Moreover, this seldom-used combination of Unix features should never result in a kernel crash. Whereas a small file system tool named FSX [8], recently released by Apple, can detect a few such anomalies, such tools have to be written by hand. Our method allows such

anomalies to be systematically uncovered using *fully automatic* techniques that thoroughly search the state space of a CCM.

4 Verifying Models Against Requirements

In Section 3 we discussed how to use CCMs to model system-call interfaces. These models may be seen as detailed specifications that implementations should adhere to. In addition, other, simpler system requirements (e.g. “a file is locked before it is accessed”) may be encoded as CCMs that act as monitors, entering bad states when an undesired system state is entered. Finally, detailed code-level CCMs can be used faithfully to model the actual behavior of a system call implementation, and can be obtained via compilation.

Given these different levels of models, one would like to check that they are in agreement, namely, that man-page models agree with requirements, that code models satisfy requirements, and that code models match man-page models. A traditional approach to handling this question involves the use of a refinement relation to check whether a given CCM refines (is faithful to) another CCM. Our methodology uses a novel, albeit math-

ematically equivalent, alternative that relies on the use of *instrumented CCMs*.

The basic idea behind this approach is the following. Given a high-level CCM (e.g., man-page model) and a lower-level one (e.g., code-level model), we use the former to track the execution of the latter. The state space of the resulting instrumented CCM is then explored to see if the wrapper ever enters a bad state (i.e., raises an unintended exception); if so, an error trace leading from the start state of the instrumented CCM is reported to the user for debugging purposes. Essentially, the instrumented CCM checks that the CCM in question *refines* its specification. The modularity property of refinement checking allows such checks to be performed at the level of component CCMs.

In order for the instrumented-CCM approach to verification to be practical, the *state-explosion* problem must be overcome: the number of states to in the instrumented CCM is likely to be intractably large. One approach to coping with state explosion is to use *abstractions* to eliminate distinctions between data values and thus reduce the number of distinct states. We are investigating using a combination of data and predicate abstraction [2] to obtain good abstractions.

A well-known drawback of abstraction-based techniques is the *false-positive* problem: a path to an error state may exist in the abstracted system that is not possible in the concrete system, owing to the loss of too much information in the abstracted conditional statements. To combat this problem our method uses counter-examples generated during reachability analysis to successively refine the data abstractions used: see the directed edge from “Instrumented CCMs” to “Error” in Figure 1. If an error trace is detected in the abstracted CCM, the trace is replayed on the unabstracted CCM to see if it is feasible. If not, conditions on transitions are modified to refine the abstracted CCM, increasing the size of its state space but eliminating the possibility of the spurious trace. We are developing an efficient algorithm for symbolically checking the feasibility of an error path in the instrumented CCM and returning the corresponding abstraction predicates if the path is not feasible.

We are also investigating the problem of constructing a *minimal* instrumented CCM for a given specification CCM and a corresponding implementation CCM. The smaller the wrapper (instrumentation), the smaller the overhead incurred during monitoring and verification. Once a property has been verified, its wrapper can be removed from the instrumented CCM and code, also leading to reduced overhead.

5 Status

We are currently developing tool support for the CCM modeling formalism with the goal of applying HCOS techniques to several variants of Linux, including SMP, Beowulf, and embedded Linux. To date, a prototype has been implemented that consists of a visual front-end for interactive specification using CCMs, and automatic generation of Java code for most CCM features. Other tools developed for the analysis of non-CCM operational models are also being retargeted to CCMs.

References

- [1] R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. In *Computer Aided Verification, 12th International Conference*, LNCS 1855, pages 280–295. Springer, 2000.
- [2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 2001, SIGPLAN Notices 36(5)*, pages 203–213, 2001.
- [3] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of 22nd International Conference on Software Engineering*, pages 439–448, 2000.
- [4] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP 2001)*, Chateau Lake Louise, Banff, Canada, October 2001. ACM SIGOPS.
- [5] R. Grosu, Y.A. Liu, S.A. Smolka, S.D. Stoller, and J. Yan. Automated software engineering using concurrent class machines. In *Proceedings of ASE’01, the 16th IEEE International Conference on Automated Software Engineering*. IEEE, 2001.
- [6] A. Kolawa and A. Hicken. Insure++: A Tool to Support Total Quality Software. <http://www.parasoft.com/insure/papers/tech.htm>, March 2001.
- [7] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. ERASER: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
- [8] A. Tevanian and C. Minshall. File System Exerciser. <http://www.codemonkey.org.uk/cruft/fsx-linux.c>, 1991.

Increasing Smart Card Dependability

Ludovic CASSET, Jean-Louis LANET

Gemplus Research Laboratory, Av du Pic de Bertagne,

13881 Gémenos cedex BP 100.

Ludovic.casset@gemplus.com; jean-louis.lanet@gemplus.com

1. Introduction

Open smart cards like Java Card provide application developers an opportunity to develop rapidly applications by offering the possibility to download during post issuance application into the card. The main drawback with this kind of smart cards is the risk to download a hostile application that may exploit a faulty implementation module of the platform. Security is always a big concern for smart cards, but the issue is getting more intense with multi-applicative platforms, post issuance code downloading and the constant growth of the complexity. Allowing post issuance need to verify that the incoming applet respects the semantics of Java byte code. For this purpose a byte code verifier checks the code during load phase. Unfortunately due to the lack of smart card resources this piece of code has not yet been implemented on a smart card.

As the correct design and implementation of the system is the key to shun an attack, we think that the byte code verifier as a key point of the Java Card security, need to be developed with formal techniques. Smart cards can gain benefits in using formal methods by improving its dependability despite the increasing complexity of this kind of system [Lan-00]. However to use formal methods in industry we need to provide a methodology to integrate it into the software development cycle. With this methodology introduced in section three, we will be able to provide metrics and figures on formal developments that can help industrials to get confidence in formal methods.

We have achieved two challenges: to embed a Java Card byte code verifier into a smart card and to develop this verifier using formal methods. Leroy has already [Ler-01], [Ler-02] developed a prototype of an embedded byte code verifier. He has also provided some formal specification of his verifier. But we are the first to have developed a byte code verifier for Java Card with formal methods. We will see in section two that implementing a byte code verifier for a smart card is not obvious and requires some improvements.

This work is a part of the European project MATISSE¹. The approach of the MATISSE project is to exploit and enhance existing generic methodologies and associated technologies that support the correct construction of software-based systems. In particular, a strong emphasis is placed on the use of the B Method [Abr-96]. Within this project we evaluate the advantages and the drawbacks of using formal methods in our specific domain.

2. Enhancing the Java Card security

For the Java Card security, it is important that an applet can not have access to the data of other applets by using the sharing mechanism, or access to the code of the operating system. The verifier examines incoming code in order to ensure that it respects the syntax of the byte code language and the language typing rules. The verifier checks statically that the control flow and the data flow do not generate run time error. Other components are responsible for protecting system resources from abuse but they depend on the verifier as they rely on language features such as access restrictions (private,

¹ European IST Project MATISSE *IST-1999-11435*

protected, final, etc). It is obvious to say that a vulnerability in this component will be catastrophic for the card. We have specified and implemented such a verifier with all the Java Card byte code features except the subroutine treatment.

We use the proof carrying code (PCC) technique to perform the on-card verification [nec-97, Ros-98, Cas-02]. This verifier scheme is similar to the KVM verifier or to the Java lightweight verification [Sun-00]. The idea is to separate the verification process in two parts as shown below. An off-card part, that computes a certificate, or “proof” indicating that the code is correct with respect to the security policy and an on-card part, that uses the certificate to verify the correctness of downloaded code.

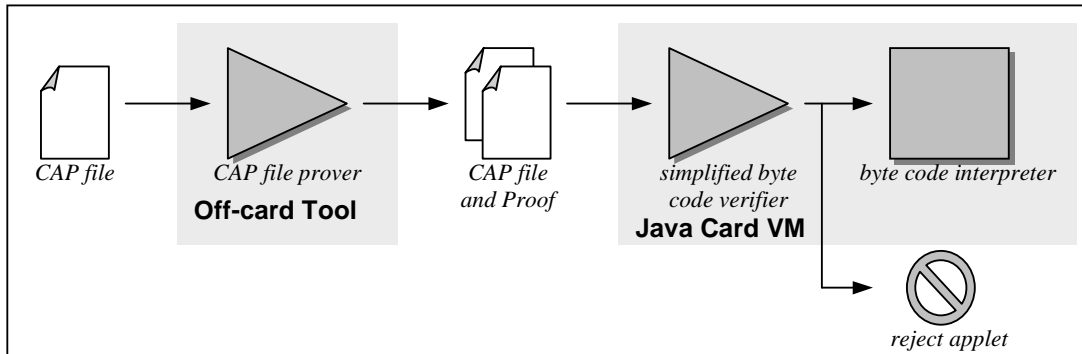


Figure 1 The PCC verifier

The “proof” generated is similar to the `StackMap` attribute used by the KVM, and contains the same kind of information. In our development, we add it in an additional component to the CAP file. The proof is built from the CAP file and the export files that have been used to build the CAP file and then added to the CAP. If the CAP file is not a valid CAP file, it is rejected and no proof is added. As shown previously, the proof added consists in type information for specific parts of the code (jump target). This proof is used later by the card to perform verification of Java Card applets when the applet is uploaded to the card. It is not needed afterwards. As this is an off-card treatment, the generation can be a memory and computation intensive process. A first part of the proof generation consists in classical byte code verification. Then, the proof is extracted from the information computed during the verification. Then the on-card verifier uses this additional information to speed up the verification algorithm which becomes linear.

At the end, the software has been embedded into smart card. The chip target is an ATMEL platform, the AT90 SC 6464C. This chip contains 64 kb for the program and 64 kb for the data and 3 kb of scratch memory. The code is stored in the program area while downloaded applet to be verified are stored in the data area.

3. The formal development from high level specification to the implementation

The formal development is included into the general methodology depicted in the following figure. This general methodology helps us identify the different step in an industrial software development, from the earlier requirements and design to the final implementation produced.

The formal development is split in three different phases:

- the translation from the informal specifications into a B model, called *formalization* on the previous figure,
- the formal development to obtain a formal implementation in B0, a subset of the B language, called the *refinement* in the next figure,
- the translation of this formal implementation into a classical programming language such as C, the *translation* phase in the previous figure.

The validation of the development is performed in several ways. As a formal development, it mainly relies on the proof activity that ensures the consistency between the formal implementation and the formal specification. Mathematical lemmas are generated at each step of the refinement process. If it is impossible to prove them, then the model is not consistent and needs to be corrected. The origin can be an error within the model, a lack of properties or of invariant. The proof process aims to prove all these lemmas and to make the correction if needed. In our example, 29 errors have been

discovered and corrected during the proof. It appears that these 29 errors have generated hundreds of non provable lemmas. Therefore, the proof phase is a powerful debugger to find and to identify errors.

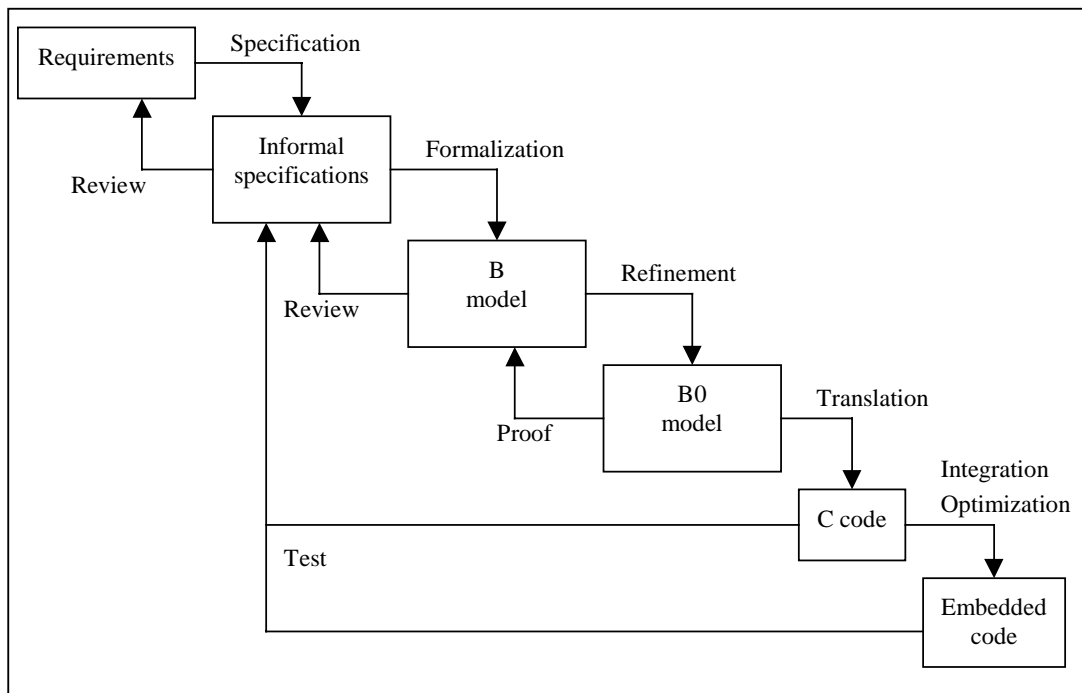


Figure 2 The software development methodology integrating formal development

But, as the translation phases, *i.e.* the translation of the informal requirements into formal models and the translation of the formal implementation into C code, can introduce errors due to their human and non formal activities, there is a need to perform some checks. Those checks are done in two different ways. The first one is to do a specification review, identified by the *review* item on the previous figure. The review may have two advantages: to be confident with the model before starting the proof phase and to reduce the possible errors introduced during the translation from the informal requirements to the formal models. In our example, the review process helped us to identify a particular error that has affected nearly 12 different instructions. This error finds its origin in the informal requirements itself, showing that without a review we could have never found this error. The review should be performed by a team different from the development in order to increase its efficiency.

The second way of completing the proof is to test the code produced. It is identified by the *test* item in the previous figure. Formal methods aim to reduce the use of test and mainly of unitary tests. However there is still a need for functional tests. That the kind of tests we aim to perform in order to ensure that the code obtained and installed on our smart card is compliant with the informal requirements. This test plan was generated by a team different from the formal development team. It was developed according to the informal requirements. It then allows us to tests the translation from the informal requirements to formal models, the translation into C code and eventual optimizations that can be performed. All this parts are identified on Figure 2. In our example, we have discovered 23 errors during testing that can be classified in two different categories. The first one is linked with errors introduced during the informal to formal translation: 14 errors have been discovered. This kind of errors are generally not detectable by the proof process. The second one is linked with tools that perform the translation from the formal implementation into a programming language. These tools are proprietary prototypes and have not been qualified according to the standard process. The number of errors discovered in this category is 9. In [Cas-02], the author compares the formal development of a byte code verifier with a conventional one, aiming to find the difference and to emphasize the benefits of using formal methods in industrial developments.

4. Conclusions

This work provides the evidence that the use of formal method for the development of an operating system in a strongly constrained device can improve the quality at an affordable cost. This technique is the most promising one for such dependable devices. Increasing the dependability of smart card is the key to allow applet post issuance downloading. Dependability can be obtained through different techniques:

- fault tolerance, which is not compatible with the smart card constraints,
- fault removal, that has shown its limits with the complexity of the new operating system,
- fault avoidance, which can drastically improve the security of this device but unfortunately it is difficult to convince managers to use it.

The results obtained with a dual development, allow us to collect metrics in accordance to an evaluation plan and to develop a methodology. We demonstrate that generating code for a smart card is possible without a too important overhead and that some choices must be carefully done during the development by identifying which part must be formalized and which one can be developed traditionally. For example, some low-level modules of the structural verifier are entirely developed with B, requiring for the proof process significant efforts. Those modules could have been developed with the standard development procedure without reducing the confidence in the code. But integrating legacy code with formally developed one is easy. With such a method it is possible to replace unitary testing by proofs that provide us a higher confidence in our code.

We also learn that the formalization of the informal specification is a key step where we have to pay a special attention. It is also a powerful means to find ambiguities in the informal specification. Most of those conclusions are well known by the community and we just point them out in our specific domain, the smart card. We provide a non trivial example of the use of formal methods. Moreover we reach our challenge, to formally implement a complex piece of code into a smart card.

References

- [Abr-96] J.R. Abrial, *The B Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.
- [Cas-02] L. Casset, *Development of an Embedded Verifier for Java Card Byte Code using Formal Methods*, In Proceedings of FME 2002, Copenhagen, Denmark, July 2002.
- [Lan-00] J. -L. Lanet, *Are Smart Cards the Ideal Domain for Applying Formal Methods ?*, In Proceedings of the ZB 2000 Conference, York, United Kingdom, September 2000.
- [Ler-01] X. Leroy, *On-Card Byte Code Verification for Java Card*, Proceedings of e-Smart, Cannes, France, September 2001.
- [Ler-02] X. Leroy, *Bytecode Verification on Java smart Cards*, to appear in Software Practice and Experience, 2002.
- [Nec-97] G. Necula, P. Lee, Proof-Carrying Code, in 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 106-119, Paris, France, 1997.
- [Ros-98] E. Rose, K. H. Rose, Lightweight Bytecode Verification, in Formal Underpinnings of Java, OOPSLA'98 Workshop, Vancouver, Canada, October. 1998.
- [Sun-00] *Connected, Limited Device Configuration*, Specification 1.0a, Java 2 Platform Micro Edition, Sun Microsystems, 2000.

A Utility-Centered Approach to Building Dependable Infrastructure Services

George Candea and Armando Fox

Stanford University, Gates 452

Stanford, CA 94305, U.S.A.

{candea,fox}@cs.stanford.edu

Abstract

Achieving dependability in large scale infrastructure systems always requires making intelligent tradeoffs. This paper draws upon ideas from economics and operations research to propose a systematic approach to thinking about such tradeoffs in terms of the system beneficiary's utility. The design process consists of choosing a spanning set of axes for the design space, explicitly formulating utility functions with respect to each axis of the spanning set, and then iteratively converging on the design that maximizes overall utility. We apply this process to the design of a fictitious online banking system.

1. Introduction

In this paper we describe a process for designing infrastructure services, particularly those that are Internet-based. Examples of such services include airline reservations (Sabre), e-commerce (Amazon), wide area caching (Akamai), searching (Google), portals (Yahoo), instant messaging (AOL), news sites (CNN), web e-mail (Hotmail), online auctions (EBay), etc.

We view dependability of a service as an expression of how well the system's properties match the system's requirements. Successful infrastructure services require significant amount of functionality, maximum correctness, must meet usability and maintainability requirements, be performant, secure, highly available, inexpensive to develop, etc. Decades of computer engineering have demonstrated the difficulty of simultaneously achieving all these properties; therefore, making smart engineering tradeoffs is vital to dependability. Paraphrasing [15], we define the dependability of an infrastructure service to be the degree in which a match between *required* and *provided* levels of availability, reliability, safety, and security has been achieved.

The importance of properly reconciling service quality, availability, performance, security, etc. increases commensurately with the system's scale. In small systems, achiev-

ing the right "mix" is typically an optimization, whereas in giant scale systems [5], sound tradeoffs become indispensable to the very possibility of building these systems. For example, airline seat reservation systems, faced with the practical impossibility of detecting duplicate bookings in real time, delegate such analysis to offline or off-peak hours, trading consistency for large transaction volume and high availability. RAID-5 systems, when scaled to hundreds of disks, have an increased probability of experiencing multiple simultaneous disk failures, which make the entire storage system fail. This led to the invention of RAID-6, which trades performance for the ability of tolerating double failures, hence decreasing the probability of data loss by 2-3 orders of magnitude [7].

There is extensive literature dealing with pairwise tradeoffs, but in practice tradeoffs are always made along more than two axes at any given time. For example, the Inktomi search engine allows for incomplete results to be returned in response to a search query, in order to obtain in exchange higher performance, higher availability, and decreased system cost. Akamai's content distribution network has cache nodes distributed worldwide to improve its level of performance, availability, and cost, but in exchange the system trades security and manageability. Finally, when Yahoo chose to implement its own hash table-like data store, instead of using an off-the-shelf database, it traded cost and portability for higher performance and more appropriate functionality. All these tradeoffs move the designed system closer to the service's requirements; without these design choices, the services would likely not have survived.

Software engineers are well aware of multiway tradeoffs employed in building computer systems, but they seldomly make these tradeoffs explicit. Consequently, building dependable infrastructure services is still an art. This paper proposes a way to bring this art one step closer to engineering, through the definition of:

- a simple model for the space in which design tradeoffs are made,
- a simple, comprehensive vocabulary for describing properties that result from these tradeoffs, and

- a step-by-step process for trading system properties against each other, such that overall system utility is maximized.

2. The Design Process

System properties can usually be described in terms of a small set of “design axes”. Some of these are rather universal, like data consistency, performance, availability, while others may be application-specific, like data lifetime, security, and interactivity. The process of designing a system amounts to attempting to maximize the overall utility of the system with respect to these properties, which we can envision as axes of a design space. We employ the utility function concept, as used in economics, to model the level of “happiness” that the beneficiary of an infrastructure system derives from different levels of the system’s properties.

Consider the following process:

1. Identify a coordinate system for the design space, i.e., a set of axes that span the design space. The notion of spanning set is used loosely to mean that any interesting tradeoff can be expressed in terms of the axes in the spanning set. Moreover, the axes need to be orthogonal, i.e., we cannot express one of them as a combination of the other axes. For example, security is orthogonal to performance and availability, whereas availability is not orthogonal to time-to-fail and time-to-repair.
2. Formulate what is typically called a “requirements specification” in terms of these axes, usually the result of discussions between the client and the system vendor. Specifically, articulate utility functions with respect to each of the spanning axes, expressing how useful a given level of that property might be.
3. Identify major design regions within the design space, akin to equivalence classes in design space, in which all designs have a common pattern. For example, “three-tiered Internet service architecture” would denote one such design region. For each design region, choose an exponent consisting of a representative design.
4. For each region exponent, find its coordinates (or ranges of coordinates) in design space. Based on these coordinates and the utility functions, compute the overall utility of that exponent. The implication is that the utility of the exponent is representative of the utilities of all designs in that region.
5. Choose the design region whose exponent has the highest overall utility. If the design is sufficiently specific, proceed to build it. Otherwise, go back to step 3

and choose subregions of the chosen design region and drill down into more detail.

To illustrate this process, in the sections that follow we present a mock design process for an online bank. Our treatment is more qualitative than quantitative; the use of specific numbers does not imply rigorousness. Our main purpose is to provide the intuition for the proposed process, rather than advocate specific axes or utility functions for the chosen application domain.

3. The Coordinate System

For the banking application we choose five axes: quality of data, service availability, performance, security, and total cost of ownership.

Quality of data reflects how “good” that data is to the user application. While generally this axis would incorporate both the notion of consistency and fidelity, for our simplified banking example we look only at consistency between displayed results and the golden copy of the account, stored in the bank’s database. The quality axis is continuous in nature and ranges from 0% to 100%.

Availability captures the percentage of read and write requests that are completed satisfactorily by the service over the lifetime of that service, i.e., the probability that a given request will be satisfactorily answered [9]. The availability axis is continuous in nature and ranges from 0% to 100%. Some services measure availability as the percentage of time they are available to reply to requests, regardless of whether such requests are issued or not; we believe such a workload-independent definition is inaccurate.

Security usually encompasses authentication, authorization, confidentiality, integrity, and accountability, which can be treated separately, if needed. In this example, we will take the approach of using an ISO standard, ITSEC [1], which takes all aspects of security into account when evaluating system security. The ITSEC, likely the most successful computer security evaluation system, was developed after the Orange Book and is closer to today’s technology. There are 7 ITSEC evaluation assurance levels (EAL). For example, EAL 3 corresponds to a system methodically tested and checked for security vulnerabilities, with grey box testing and selective independent confirmation of developer test results. The toughest level, EAL 7, requires that a system’s design be formally verified and tested, with the formal model supplemented by a formal presentation of the functional specification and high level design, showing correspondence between design and implementation. There exist a variety of certified commercial products, such as the Cisco Secure PIX Firewall 5.2 (EAL 4), Oracle 8i (EAL 4), Sun Solaris 8 (EAL 4), Hitachi MULT-OS v3 (EAL 6), etc. The security axis is discrete, taking on EAL values from 0 to 7.

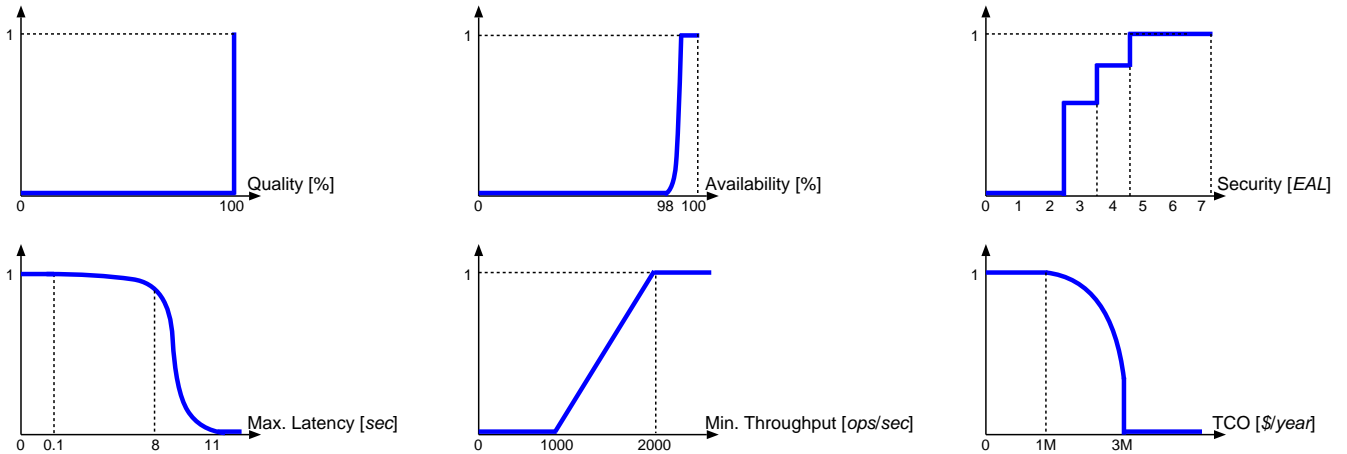


Figure 1. Normalized utility functions for an example online banking application.

Other possible quantifications of security include a simple set containment approach, in which higher levels of security incorporate a larger set of security precautions included in the system. Another approach, depending on the application, is to quantify security according to cryptographic key sizes [17]. [20] proposes a way to evaluate and quantify the security of storage systems.

Performance is usually viewed as an expression of the throughput and latency of access. For a general banking service we may want to consider both read and update throughput/latency and perhaps even differentiate based on the particular data set being accessed. However, in this example we will only look at a general measure of overall throughput and latency. The performance axes have a continuous value set; throughput is measured in operations/second, and latency in seconds.

Total cost of ownership (TCO) includes hardware/software costs, training, maintenance, technical support, network connectivity, etc. In this example we use the per-year amortized cost, with the TCO axis quantified in dollars/year.

When choosing values and metrics for points on any of the design axes, system designers will generally choose units specific to the applications that will use the state repository they are building.

4. The Requirements Specification

The requirements specification is a collection of utility functions, one for each axis of the design space, along with a formula for combining individual utilities into an overall utility. It is acceptable for points on the axes to not be quantified with absolute metrics; what really matters is that values can be compared to each other. The units used for measuring utility need to be uniform across all five axes, to

be able to correctly compare utilities throughout the design space.

Utility functions can be specified at various levels of detail, from qualitative graphs to precise quantitative functions. The right level of detail is generally obtained at the end of an iterative process, in which utility functions are successively refined. The general approach we propose for building these utility functions is to choose salient points and then qualitatively interpolate between them.

In this example, the formula for combining utilities is simply a multiplication.

Quality: Certain applications, such as large search engines, routinely reduce completeness of their answers [5], that, however, would be unacceptable in the case of a banking application, where consistency between the reported balances/payments/etc. with the true bank account is crucial. Therefore, the only salient point in this case is the 100% quality point, and Figure 1 shows one of the simplest utility curves possible: a step function. Any quality below 100% is useless, hence utility 0; once the quality is 100%, it perfectly meets the requirement of the application.

Availability of the service is the percentage of requests that are satisfactorily fulfilled by the bank's web site. According to various surveys, the true availability of the best-of-breed web sites today is on the order of 98%, so we choose that as one salient point. For competitive reasons, we would expect the online bank to find a system with poorer availability than 98% to be totally useless. The utility of availability rapidly increases until it reaches the order of 3 nines, after which any further improvements in availability become rather worthless, as we can count on users to retry a failed request. This yields a second salient point at 99.9%. We interpolate and show the resulting curve in Figure 1.

Security of the service is defined in terms of how useful the different assurance levels would be, so it is natural to

choose the assurance levels as salient points. We could say any security less than EAL 3 would be considered inappropriate for a bank, while an EAL of 5 or above is plentiful—beyond a certain point, other factors become of greater concern than the security of the service itself (e.g., disgruntled employees, administrator mistakes, etc.) The levels of 3, 4, and 5 naturally correspond to different levels of utility.

Performance of the service is described by latency and throughput. When determining the salient points for latency, we can resort to research that looks at how web response times affect the user experience. For example, [2] shows that the web site performance threshold at which customers get frustrated and leave is between 8-11 seconds. Another study [4] shows that response times of 100 msec or less give the feeling of instantaneous response, hence any response time between 0-100 msec should have utility 1. We interpolate between these points and obtain the latency utility curve shown in Figure 1. For throughput, in our example, we arbitrarily decide that a value below 1,000 operations/second is worthless and any throughput of 2,000 or more operations/second is sufficient. To interpolate between these two salient points we use the observation in [2] that there is a linear 50 percent relationship between site performance and site abandonment.

Generating a utility function for *total cost of ownership* is a highly non-technical task. Assume, for the sake of illustration, that the approved IT budget for this project is \$1M for each year of operation, and the total IT budget for all of the bank's operations is \$3M/year. Therefore, as long as TCO stays below \$1M, the utility of the resulting system is maximized with respect to cost. It is possible to stretch this cost up to \$3M (at decreasing utility), beyond which it is impossible to support the system.

5. Finding A Global Maximum

In this section we walk through the iterative process of converging toward a design that maximizes overall utility. We repeatedly identify regions of the design space and compute their expected range of overall utility. In the first phase, there is a wide variety of representative implementations for the type of services described here; we choose two of them for illustration.

System 1 is a completely distributed design. The US customer base is served by five geographically distributed clusters of web servers. Clients are routed to their nearest front end via DNS mappings. Behind these front ends lie application servers implementing the bank's business logic; each application server converses with a local replica of a distributed database. Any time updates are made, they are geographically distributed to all replicas using some efficient mechanism, such as log-based replication.

System 2 is a totally centralized design. One single data center serves all customers, a single database holds all the

account information, and there is a single cluster of application server instances in the middle tier. A load balancer distributes requests to the web servers in the front end, that communicate with the application server.

In the table below we estimate in each column the utility we would get from each system along each axis of the design coordinate system. For example, the quality for both systems, since they are implemented on top of ACID databases, is always 100%, hence a utility of 1. Security utility for the first type of system is 0, because one cannot presently assemble such a distributed system from software components certified at EAL 3 or better. TCO is better for the second system because of it being centralized and thus easier to manage. The numbers shown are somewhat arbitrary—we chose them to illustrate distinguishing features, rather than prescribe certain utility values for online banks.

Region	Quality	Availability	Security	Latency	Throughput	TCO	Overall
1	1.0	0.9 - 1.0	0	0.9 - 1.0	0.9 - 1.0	0.5 - 0.7	0
2	1.0	0.2 - 0.4	0 - 1.0	0.8 - 0.9	0.6 - 0.8	0.7 - 0.9	0 - 0.26

To compute the overall utility range, we multiply the individual utilities. Based on the results, we choose the design region that contains totally centralized systems and proceed to the second phase. In the chosen region, we refine the *System 2* type into two other representative design points:

System 2.1: For the data center we choose a Sun Solaris 8 platform, running the Oracle 8i database server in conjunction with BEA's WebLogic 7.0 application server. The front ends are Netscape Enterprise 3.6 web servers.

System 2.2: The platform is Redhat Linux 7.2. We hire an outside company to implement a custom database that offers specific performance and functionality properties not found in commercial databases. The middle tier is entirely developed in house, and the web front ends consist of Apache 2.0 web servers.

Similar to the first phase, we construct a table with the axis-specific utilities of the two possible choices.

System	Quality	Availability	Security	Latency	Throughput	TCO	Overall
2.1	1.0	0.2 - 0.4	0.5 - 1.0	0.8	0.8	0.7 - 0.8	0.05 - 0.21
2.2	1.0	0.3 - 0.4	0 - 0.5	0.9	0.8	0.8 - 0.9	0 - 0.13

Based on these estimates, we choose the first option, *System 2.1*. We only showed these first two phases, but the process would not stop here—it would proceed with further refinement of the chosen system, including various configuration aspects, choices for finer grain components, etc.

6. Discussion

The example presented here is very simplistic, as our intention was to illustrate a way of thinking, rather than

give specifics of the online banking system. For clarity, we normalized all utility functions and assumed that they are equally important to the final computation. When this is not the case, one can simply scale each utility function appropriately (e.g., if security is more important than other properties, its maximum utility could be 5 instead of 1). Alternatively, we can give different weights to the utilities in the combining formula.

6.1. The Design Hyperspace

The six axes used in the above example form a 6-dimensional space; each possible implementation of the desired system corresponds to a point in this space. For each point in this space, we can compute its overall utility. If we consider utility as a seventh axis for this design space, then the implementations with their utilities describe a discrete “utility manifold” in 7-dimensional space. Making tradeoffs consists of navigating this manifold in search for the global utility maximum, a point at which tradeoffs are optimal given the utility functions. As utility functions change (varying user demands, market pressures, etc.), the utility manifold changes in shape; as technology changes, new points may appear or disappear in the design space.

It is difficult to reason in terms of “navigating” a manifold in 7-dimensional space, which is why designers reason mostly in terms of pairwise tradeoffs. However, any one property often affects two or more of the other properties, so the overall design process does take place in this 7-dimensional space—a fact that must be recognized and incorporated in our software development methods.

6.2. Global Plateaus, Not Maxima

In the process shown above, we made the simplifying assumption that the utility manifold is smooth, i.e., no “cliffs” are encountered when moving from one point to another. Such cliffs do exist, however [10]: for example, when a front end node is hit with high traffic, there is a point at which it starts thrashing, causing performance to drop all of a sudden. The true aim of the design process is therefore not an absolute global maximum, but rather a global plateau, which provides both a high overall utility and a high tolerance to disturbances.

If such cliffs are present within a region, choosing an exponent with a representative utility for the region is more difficult—it may become necessary to choose smaller regions or evaluate multiple exponents for each region. The number of phases in the design process is proportional to the smoothness of the utility manifold: the fewer cliffs, the fewer phases. Moreover, the shape of the utility manifold in the neighborhood of a chosen point can reveal some valuable tradeoffs that were overlooked, thus guiding the designer in choosing neighboring points to explore.

6.3. Dynamic Runtime Tradeoffs

Unpredictable workload is the norm in large scale infrastructures, and over-provisioning to handle all possible load spikes is most of the time too costly [3]; fast dynamic tradeoffs are therefore required. Some, like the public telephone system, trade availability for quality by blocking the initiation of calls in overload situations. Others, such as CNN.com, reduce richness of web pages to keep availability constant during high load periods [16]. We believe our utility-based approach is well-suited to building adaptable systems that make tradeoffs at runtime, e.g., by changing operating parameters. Operators can express requirements through the utility functions, and the system autonomously changes system parameters to maximize utility, without requiring human anticipation or authorization of the tradeoffs. A challenge in applying this method to machine-chosen tradeoffs is the need to have concrete metrics for the axes and explicit utility functions.

6.4. Utility Functions Are Hard to Formulate

The design process is complex and almost always includes multiple interactions with the target system’s beneficiary, to understand what the requirements really are. It is difficult to make the requirements explicit, and often clients themselves do not understand their intended use of the system. For example, consultants at Oracle Corp. are often asked by clients to build a data warehouse that is available 24x7. In many cases, however, it turns out that the client’s updates and accesses will be run in batch mode, and so the usefulness of 24x7 availability is not much higher than a somewhat more reduced level, yet the cost of taking the data warehouse to that level is significant [19]. Luckily, stating utility functions is easier for systems that both the clients and the providers have had previous experience with.

Even if many clients do not know what their utility functions are, these functions do exist. They need to be made explicit, if the resulting system is to be dependable. As suggested in [13], well-designed graphical tools can guide users in enunciating requirements, as well as provide “what-if” analyses to confirm the functions are valid.

6.5. Related Work

In choosing economic tools and concepts to represent customer requirements, we were inspired by the architecture for Internet service levels proposed by [21] as well as by the market-based approach to resource allocation described in [11]. The field of operations research has already developed an extensive theory [12] on the use of utilities in making decisions and value tradeoffs, which we intend to peruse in our future work. Value analysis [18] is a

well-established engineering technique that provides a disciplined, step-by-step approach to identifying and removing unnecessary cost in product and service design—a similar approach to what we are trying to develop.

Hippodrome [3] employed an iterative design process to configure storage systems, similar in spirit to what we described here. [6] used a utility function approach for energy and server resources in large data centers. Extensive work has also been done in identifying and making pairwise tradeoffs in systems; two notable examples include Bayou [8] and TACT [22]. The idea of exploring global maxima in a design landscape forms the basis of genetic programming [14]; in maximizing a fitness function (rather than an overall utility function), genetic algorithms use the principles of Darwinian natural selection to converge onto an optimal solution.

7. Conclusion

In this paper we argued that tradeoffs in computer system design always take place in a multidimensional space, rather than just along two axes. Understanding this fact and having a suitable model is particularly important in large scale infrastructure services, where the right tradeoffs are critical to the very existence of the service. We described an iterative, high level process based on utility functions that can help in better matching system properties to the beneficiary's requirements, hence improving system dependability. We illustrated this process with a simple example of choosing the right commercial software for a banking service; the same process also applies to the development of code.

8. Acknowledgements

We would like to thank the Stanford/Berkeley Recovery Oriented Computing team and Eric Anderson, Dan Candea, Andy Chou, Kevin Fu, James Hamilton, Kim Keeton, Cost Sapuntzakis, Alfred Spector, Marvin Theimer, and Amin Vahdat for valuable insights and comments on the material presented here.

References

- [1] Information technology security evaluation criteria. ISO-15408.
- [2] Tying performance to profit. Technical Report STP01-C04, Jupiter Media Metrix (Jupiter Research), New York, NY, June 2001.
- [3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the Conference on File and Storage Technologies (FAST-02)*, pages 175–188, Monterey, CA, 2002.
- [4] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *Ninth International World Wide Web Conference (WWW9)*, Amsterdam, The Netherlands, May 2000.
- [5] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July 2001.
- [6] J. Chase, D. Anderson, P. Thakar, and A. Vahdat. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 103–116, Banff, Canada, 2001.
- [7] P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.
- [8] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the 1994 Workshop on Mobile Computing Systems and Applications*, December 1994.
- [9] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, San Francisco, CA, 1993.
- [10] S. D. Gribble. Robustness in complex systems. In *Eighth Intl. Symposium on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau, Germany, May 2001.
- [11] K. Harty and D. Cheriton. A market approach to operating system memory allocation. In *Market-Based Control: A Paradigm for Distributed Resource Allocation*, Singapore, 1996. World Scientific Publishing Co.
- [12] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. John Wiley & Sons, 1976.
- [13] K. Keeton and J. Wilkes. Automating data dependability. In *Proceedings of the SIGOPS European Workshop (2002)*, Saint-Emilion, France, Sep 2002.
- [14] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.
- [15] J.-C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, Vienna, Austria, Dec 1991.
- [16] W. LeFebvre. CNN.com – facing a world crisis. Invited talk at USENIX LISA, San Diego, CA, Dec 2001.
- [17] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, Sep 2001.
- [18] L. D. Miles. *Techniques of Value Analysis and Engineering*. McGraw-Hill, 1972. 2nd edition.
- [19] M. Moy. Personal Communication, 2002.
- [20] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *Proceedings of the Conference on File and Storage Technology*, pages 15–30, Monterey, CA, Jan 2002.
- [21] S. Shenker. Fundamental design issues for the future internet. *IEEE Journal on Selected Areas in Communications*, 13(7):1176–1188, Sep 1995.
- [22] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 305–318, Oct. 2000.

Model Checking System Software with CMC

Madanlal Musuvathi*
Stanford University
madan@cs.stanford.edu

Andy Chou
Stanford University

David L. Dill
Stanford University

Dawson Engler
Stanford University

Abstract

Complex systems have errors that involve mishandled corner cases in intricate sequences of events. Conventional testing techniques usually miss these errors. In recent years, formal verification techniques such as [5] have gained popularity in checking a property in all possible behaviors of a system. However, such techniques involve generating an abstract model of the system. Such an abstraction process is unreliable, difficult and miss a lot of implementation errors.

CMC is a framework for model checking a broad class of software written in the C programming language. CMC runs the software implementation directly without deriving an abstract model of the code. We used CMC to model check an existing implementation of AODV (Ad Hoc On Demand Distance Vector) routing protocol and found a total of 29 bugs in two implementations [7],[6] of the protocol. One of them is a bug in the actual specification of the AODV protocol [3]. We also used CMC on the IP Fragmentation module in the Linux TCP/IPv4 stack and verified its correctness for up to 4 fragments per packet.

1 Introduction

The reliability of system software is particularly important in areas such as communication protocols, security-sensitive applications, and embedded software. In these applications, failures may affect many machines, compromise the integrity or privacy of user data, subvert the ability of an administrator to control the system, or totally cripple the application. On the other hand, errors in these systems are difficult to find as they usually involve mishandled corner cases triggered by intricate sequences of events. Conventional testing techniques are not able to detect these errors for two key reasons. Firstly, traditional coverage techniques [1] do not guarantee coverage across all sequences of events. Secondly, the occurrence of systemic events such as packet loss, link failure or disk crashes are not easily controlled in a test environment. Thus, even a rigorously tested system contains a residue of errors that either cause the system to crash after

long periods of execution, or present a potential security hole to a malicious user.

In recent years model checking [2] has gained popularity as an automatic verification technique that enumerates, either explicitly or implicitly, all possible states of a finite state system. Conventional model checkers [5] usually assume that the design is described at a high level that abstracts away many implementation details. Verifying actual code using such a tool requires reconstructing this abstract description from the code. This process requires a great deal of manual effort. Moreover, human errors in the manual abstraction result in missing bugs and causing false alarms during the verification process. A final, serious problem is that the models are usually written in a special-purpose description language that do not support the low level semantics of C, the preferred language for system implementations. As a result the models are often so abstract that there is a significant semantic gap between such a model and an implementation. For these reasons, it is a notable curiosity when software is model checked, rather than an everyday occurrence.

This paper describes CMC, a C Model Checker. CMC is a framework for model checking broad range of software implementations directly. Given an implementation of the system, a set of events that affect the system, and the event handlers in the implementation that handle each event, CMC explores the state space of the system by systematically executing all possible sequences of events. After each event execution, it calls the *pickle* function provided by the user to extract the state of the system and stores the state in a hashtable. By remembering the states visited, CMC explores each state of the system only once. The primary advantages of this approach follow from the ability to model check an implementation directly. No model of the code needs to be constructed, refined, or debugged. In addition, we can leverage existing dynamic techniques for debugging code, such as memory leak and stack overflow detectors. These tools have exhaustive coverage when run in the CMC framework.

To our knowledge, Verisoft[4] is the only other tool that is able to model check the implementation directly. However, Verisoft does not store states. As a result, states that have been explored before are redundantly checked, and systems can not be exhaustively verified if cycles exist in the state space. Interesting systems almost always have state spaces

*Partially supported by GSRC/MARCO Grant No: SA3276JB

with cycles because otherwise only a finite number of events (e.g. message sends) can be handled before stopping.

2 Design of CMC

CMC models the system being verified as one or more *processes*. These processes execute the native code of the implementation in the context of the model checker. Processes can be separate implementations or different instances of the same implementation (e.g. multiple nodes running the same routing protocol).

Each process has a private *internal state* and uses a shared *global state* to communicate with other processes. At any instant, the *system state* is the aggregate of the internal states of the processes and the shared global state. A *transition* of the system is defined as the execution of a deterministic, atomic procedure by a process that modifies its internal state and the global state. The set of transitions for a process are exposed to CMC via a standard interface.

Figure 1 shows a stripped-down implementation of a routing protocol. A process is a node running the protocol in a network. The internal state of each process is its IP address and its routing table. The routing table is simply a linked list containing the next hop for each destination IP address. Each process has two transitions, `recv_packet()` and `on_link_failure()`, which modify the route table in response to network events. The global state of the system is the state of the network, which is not shown in the figure. The system state is the aggregate of the internal states of all of the processes in the network and the global state.

CMC model checks software using explicit state enumeration. From each system state, it executes all transitions of all processes to determine the set of successor states. Applying this recursively from the initial state, CMC explores the entire set of reachable states of the system. CMC maintains a hash table of visited states and never explores a state more than once. We use hash compaction [10] to guarantee negligibly small probability of collisions. For each visited state, a set of invariants are checked to determine the correctness of the system.

CMC requires that processes provide two functions, *pickle* and *unpickle*. A *pickle* function gathers the internal state of a process, consisting of global variables and heap data, and produces a concise representation of the state. An *unpickle* function inverts the *pickle* function, taking the state and restoring the global variables and heap data. These two functions provide access to the internal state of the processes and make state space enumeration possible.

Given these two functions, CMC executes a transition as follows. It unpickles the current state of the process and executes the transition by running the implementation. This is the only time the implementation has control of the processor. When the transition function returns, CMC pickles the

state of the process to determine the next system state. The current implementation of CMC requires that processes execute events “atomically” in the sense that they run to completion without blocking. This restriction also eliminates the state of the stack from a process’ internal state.

In Figure 1, the `pickle()` function takes the IP address and route table entries for a process and copies them into a state buffer. The `unpickle()` function does the inverse, taking a state buffer and restoring the IP address and route table entries.

Real systems pose two key challenges. First, the state space might be infinite. However, the state space can be effectively pruned using techniques such as limiting the number of processes, constraining state variables to a subrange, and limiting the size of the heap. Even after applying these techniques, the state space might still be too large to complete search the entire space. We are currently investigating various heuristic search techniques to direct the search.

Second, internal state of a process may contain complex data structures, and implementing *pickle* and *unpickle* may be difficult. However, as these functions involve a traversal of the data structures, skeleton of such functions could be generated automatically using the type definitions. We are currently exploring this possibility. In our case studies (§3.1), we wrote the *pickle* and *unpickle* functions by hand. However, we used the accessor functions in the implementation itself, which greatly simplified the process.

CMC is particularly suited to systems where the behavior of interleaving executions of multiple processes gives rise to complex, emergent behavior. As long as the state space of the model is kept small enough, CMC can be very effective at detecting unexpected behaviors. On the other hand, using CMC requires a nontrivial amount of work, which makes it difficult to apply to large amounts of code. As described above, the system also imposes restrictions that may preclude its use on certain programs.

3 Model Checking Case Studies

In this section we discuss our experience with CMC on two case studies: AODV (Ad-hoc On-demand Distance Vector)[3] routing protocol, and the IP fragmentation module in the Linux kernel (Version 2.4.18).

3.1 Verifying AODV

AODV is a loop-free distance vector protocol for ad-hoc networks [3]. We applied CMC on two implementations of the protocol. The first implementation, *mad-hoc* [7], was released two years ago and has been under active development since. It runs as a user-space module and contains approximately 5500 lines of code. The second implementation, *Kernel AODV* [6], which descends from *mad-hoc* was released a

```

/* internal state of process */
IPAddress my_ip;
struct RouteEntry {
    IPAddress dest_ip;
    IPAddress next_hop;
    struct RouteEntry* next;
} *route_table;

/* transition functions */
recv_packet(buffer, len) {
    parse buffer;
    update route_table;
}
on_link_failure(neighbor) {
    for each entry in route_table
        if entry->next_hop == neighbor
            remove entry from route_table;
}

/* pickling code */
pickle(state){
    copy my_ip into state;
    for each entry in route_table
        copy dest_ip, next_hop into state;
}
unpickle(state){
    copy my_ip from state;
    for each entry in state
        insert entry into route_table
}

```

Figure 1. A skeleton routing protocol implementation.

bit less than a year ago. It contains 7500 lines of code and runs as a loadable kernel module in Linux and ARM based PDAs.

We modeled up to 4 AODV processes, each running an instance of the same implementation. The state of the process consisted of several global variables and the routing table, implemented as a link list. The pickle and unpickle functions traverse the entries in the routing table, using the accessor functions present in the implementation. The pickle and unpickle functions were straightforward to write and are 50 lines of code each. The AODV model, including the pickle and unpickle functions were shared between the two implementations, with minor modifications.

Table 1 summarizes the set of bugs found using CMC in both AODV implementations. The bugs range from simple memory errors to protocol invariant violations. We found a total of 29 unique bugs in the two implementations. The Kernel AODV implementation has 5 bugs (shown in parenthesis in the table) that are instances of the same bug in the mad-hoc implementation. The AODV specification bug is one of them, since both implement the same specification.

We describe the bugs below at a high level to give a feel for the breadth of coverage provided by CMC.

Memory errors. The first three error classes were various ways to mishandle dynamically allocated memory: not checking for allocation failure (10 errors), not freeing allocated memory (8 errors), or using memory after freeing it (2 errors). These were all detected by the built-in memory

manager in CMC.

Both implementations carefully checked the pointer returned by *malloc* was not null. However, functions that call *malloc* indirectly can also return null pointers when the allocation fails. The code only erratically checked such cases. Since CMC directly executes the implementation, such errors were manifested in segmentation faults.

Most of the memory leaks were similarly caused by mishandled allocation failures. Commonly, code would attempt to do two memory allocations and, if the first allocation succeeded but the second failed, would return with an error, leaking the first pointer.

Unexpected messages. CMC detected two places where unexpected messages would cause mad-hoc to crash with a segmentation violation. This occurred when the system lost its state, either due to a reboot, or due to a timeout between a request response cycle. When the system received the response in its altered state, it resulted in a segmentation violation. This is a serious security violation as an attacker can maliciously send a bogus response.

Invalid messages. There were 5 cases of invalid packets being created, 3 cases of using uninitialized variables (these could not be detected by gcc -Wall), and 2 cases where invalid routes were used to send routing updates, violating the AODV specification; CMC also detected 2 instances of integer overflow which resulted in program assertion failures. Both implementations use an 8 bit integer to store the hop counts and use 255 to represent a hopcount of infinity. In the two error cases, an infinite hopcount was erroneously incremented to 0. This also accounted for a program assertion failure, as it resulted in an invalid routing table.

Routing loops. As AODV is a loop-free routing protocol, any routing loop produced during the execution of the implementation is a bug. We found three instances of routing loops, one of which is a bug in the AODV specification [3]. The two routing loop errors resulting due to implementation error are discussed here. In one, the implementation performs a sequence number comparison before a subsequent increment, while the AODV specification requires the comparison to be done after the increment: In the second case, the implementation fails to increment a sequence number while processing specific protocol message, viz. the RERR message of AODV.

The specification bug. This bug involved the handling of “route-error” (RERR) messages. In AODV, every route has a sequence number that determines the “freshness” of the route. AODV guarantees loop-freeness by appropriately manipulating these sequence numbers. When a node receives an RERR from its next hop, it sets the sequence number of its route to the sequence number in an RERR message. Under normal conditions this is the right thing to do. However, when the underlying link layer can reorder messages, the RERR message might have an outdated sequence num-

	mad-hoc AODV	Kernel AODV
Mishandling <i>malloc</i> failures	4	6
Memory Leaks	5	3
Use after free	1	1
Unexpected Message	2	0
Generating Invalid Packets	3	0 (2)
Program Assertion Failures	1	0 (1)
Routing Loops	2	1 (2)
Total	18	11 (16)

Table 1. Summary of bugs found in the two implementations of AODV

ber resulting in the node setting its sequence number to an older version. This can ultimately result in a routing loop. This bug was mentioned to the authors of the protocol with a suggested fix. Both the bug and the fix were accepted by the protocol authors[8].

The specification bug was found by running 4 AODV nodes using a depth-first search of the state space. CMC came up with an error trace of length 93. By using a best-first search we were able to find traces as short as 27. Performing a breadth-first search of the state space would give the shortest trace. However, breadth-first search on AODV ran out of resources without finding the bug. A careful hand crafted simulation of the bug required at least 25 transitions. An error of this complexity would be very difficult to catch using conventional testing means.

3.2 Verifying IP Fragmentation

We verified the IPv4 Fragmentation[9] module in the Linux Kernel (Version 2.4.18). This module assembles all fragments of an IP packet before sending it to the higher layer. This module, along with the *skbuff* library it uses, contains 1850 lines of code. In order for these kernel modules to run in user space, we provided stub functions, most of which were automatically generated. There are 21 stub functions with a total of 150 lines of code.

We verified the IP Fragmentation module with all sequences of 4 or less fragments of an IP packet. We did not find any bugs in this module. We did find a known bug in a previous version of this module, though. This case study is a proof of concept that its possible to model check kernel code using CMC.

4 Future Work

CMC is still in its preliminary stage and the results we have achieved till now are encouraging. We believe that CMC will be applicable to a wide variety of implementations. Currently we are looking into verifying properties in TCP stack implementations, filesystems, kernel schedulers, and security-sensitive applications such as root programs.

We are also looking into techniques to automate some of the steps needed to use CMC with an existing implementation. For example, compiler support might be used to help the user write the pickle and unpickle functions. Finally, we are evaluating different heuristic search techniques to guide our model checking.

5 Acknowledgments

We like to thank Satyaki Das and David Park for the numerous discussions on this topic. We also thank David Park for providing the implementation of hash compaction. We also would like to thank the blind reviewers who provided valuable suggestions to the first draft of this paper.

References

- [1] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.
- [2] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [3] C.Perkins, E. Royer, and S. Das. Ad Hoc On Demand Distance Vector (AODV) Routing. IETF Draft, <http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-10.txt>, January 2002.
- [4] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.
- [5] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
- [6] Luke Klein-Berndt and et.al. Kernel AODV Implementation. http://w3.antd.nist.gov/wctg/aodv_kernel/.
- [7] F. Lilieblad and et.al. Mad-hoc AODV Implementation. <http://mad-hoc.flyinglinux.net/>.
- [8] Charles E. Perkins, Elizabeth M. Royer, and Samir R. Das. Private Email Communication.
- [9] J. Postel. Internet Protocol. RFC 791, USC/Information Sciences Institute, September 1981.
- [10] U. Stern and D. L. Dill. A New Scheme for Memory-Efficient Probabilistic Verification. In *IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, 1996.

OASIS project: deterministic real-time for safety critical embedded systems

Stéphane Louise, Vincent David, Jean Delcoigne,
and Christophe Aussaguès

CEA-DRT/LIST/DTSI/SLA- Bâtiment 451 - CEA/Saclay
91191 Gif sur Yvette CEDEX - France
(e-mail:firstname.lastname@cea.fr)

15th March 2002

Introduction

Safety critical systems is a growing industrial concern. It is a particular and long time interest for embedded or I&C systems, in nuclear power plant or aircraft applications. Since automotive industry is to use more and more microcontrollers or microprocessors with software in the near future[Bre01], concerns about safety of these systems is becoming mainstream. At the system level, because of intrinsic complexity, it is difficult to guarantee a high dependability. Typical applications should be able to manage numerous control tasks with several different time scales. They do not only demand correctness of algorithms, but also a correct management of tight time constraints, usually dictated by the environment. For system and application design, there is a conflicting interest between algorithm design and time scheduling design. Algorithm design favors a task per targeted control and so a multitasking approach. Time scheduling, on the other hand, has to take care of both strict and precise local chain of events, that needs a careful design in a multitasking environment, and mostly independent events at the system scale, more multitasking prone. One of the highest difficulties arises from the possible existence of very different time scales.

Overview of the OASIS model

The OASIS model comes from a project led by the CEA in association with EDF and Framatome-ANP. Its main goals are to bring a solution to often met issues in the design, implementation and qualification of safety critical embedded systems. One of the main goal of the OASIS project is to allow the programmer to focus on algorithms. Simple statements of time constraints for each task environment -called agent- are enough to guarantee timeliness in the execution, as long as the system sizing is correct for the application. The key points, as shall be seen, is time and scheduling full determinism both in processing start times and deadlines, and in the communication mechanisms. It also allows a full determinism for the application behavior both when system is in nominal functioning and when it is in anomalous functioning thanks to tasks confining. A further consequence is to enable designers to check that the system fits its maximum workload, and also to demonstrate the system safety.

Functions of general I&C systems are data acquisition, processing and operating control devices. Classical system design generally leads to artificially break apart some of these aspects of a single action on a system. Secondly it leads to a multiplication of the number of tasks and communications among them. As a consequence, system worst case load is harder to compute and the application harder to design. This also leads to a higher cost of the system due to bad sizing, to a higher design and implementation cost, and complicates check and validation phases.

In the OASIS model, an application is defined as a set of a known finite number of agents (tasks) enabled to communicate if need be, and cooperating in order to reach their objectives. Each agent is an autonomous running entity composed of a determined number of elementary processing acts. These acts can be chained, depending on internal or external conditions to any agent. The whole model is based on a Time Triggered Approach[Kop] which means that commutations are triggered off by timer interrupts. Each basic processing is associated with an interval of time for its execution, determined by time related information embedded in agent program code.

Time and communication in the OASIS model

The basic mechanism for time management with OASIS is modeled with the “ADV” (advance) instruction. Let’s show the following example:

$$\{processing_1; ADV(m); processing_2; ADV(n);\}$$

this means that $start_2 = agent_start_date + m$ is the latest end date of *processing_1* and also the earliest start date of *processing_2*. $end_2 = start_2 + n$ is the the latest end date for *processing_2*, then completely defining the interval of time for *processing_2*. As a consequence, the OASIS micro-kernel ensures that *processing_2* cannot start before date $start_2$, that *processing_2* shall be run between $start_2$ and end_2 dates. Moreover, if its execution is not finished before end_2 , it stops *processing_2* then forks execution for error processing. It should be noticed that for determinism sake, the actual date, time and system state seen by any processing correspond to its earliest start date. Moreover, each agent can be associated with a different default clock so that it is easier for the programmer to manage different time scales, but all time statements can be associated with another agent clock if need be.

The OASIS model enables to use such time constraints with any control flow statements, so that elaborate processing acts chaining can be implemented, as shall be shown in the full paper.

The other important mechanism for parallel programs like OASIS applications is the availability of some kind of information sharing. There are two such mechanisms in the OASIS model. The first one is temporal variables, real time data flows. This is an information that can be shared implicitly between agents. Past values of this time stamped data flow can be read by any agent that needs it. The second mechanism for data communication is explicit message passing. Any processing of an agent can send a message to another agent, at a given date in the future. As a consequence this mechanism can impose further restrictions on interval of time available for a processing execution, but it is also taken in charge by the OASIS model and tools. It also avoids by construction any underflow for these kinds of information sharing between agents. Both mechanisms shall receive further explanations in the full paper.

Timeliness

Timeliness is the ability to execute all planned activities in a timely manner. This requires to demonstrate three properties: no processing is delayed or omitted, no processing is started too early (*i.e.*

before its earliest start date) and all processing meet their deadlines. The first two properties are verified by construction in the OASIS model and runtime environment[DAD00], as shall be clearly demonstrated in the full paper. This is a simple consequence both of the model and of the timing constraints included in the program code of each agent. Since the OASIS model is a Time Triggered Approach, a simple time-driven scheduling at runtime enables to perfectly manage processing activation and early detection of any anomalous, time related behavior of the application. As a consequence, no asynchrony can impact these basic properties of the OASIS model. The third property is then tied to the schedulability of the application and as shown in the following, to system sizing or computing power.

Sizing and schedulability

Schedulability and ability to meet every processing deadline is tied to the knowledge of an actual upper bound to all processing computing time in the worst case. Knowing the exact worst case execution time is not a necessity for schedulability. Knowing an upper bound (even large) is enough to ensure that all processing can be finished on time by using a deadline driven scheduler to trigger off tasks activation on the target systems. This technique is efficient, rigorous and optimal if programming model is adequate, like the OASIS model. With the knowledge of these upper bounds it is possible to be sure that the target system has enough computing power to run the application in the worst case. Other related works has shown how to statically compute the worst case workload of a target system that use a dynamic deadline driven scheduler. What is noticeable by comparison with other models is that the sizing issues are a completely separated matter from the application design issues (and in particular, time design issues).

Of course, knowing tighter upper bound to worst case execution time for each processing enables also a tighter evaluation of computing power needed and thus a better evaluation of system sizing. This can allow a cost reduction of the target system and is a special interest for industrial applications. That is why, although this issue is not directly related to the OASIS model, the evaluation of worst case execution time is also an interest of the OASIS project team.

The OASIS development tools

All necessary tools for implementation of the OASIS model for an embedded application, have been developed and are already fully operational. The code production chain tool include a compiler for a semi-formal language ΨC implementing the basic OASIS mechanisms (ADV, temporal variables and message communication) where purely algorithmic parts are written in *ANSI - C*. The compiler parses and checks consistency of the code as it generates a neutral code. The whole code generation also generates a complete task interface runtime, compiles and links all the code to obtain the complete application. The linking stage also generates static memory tables to protect the whole application and the runtime so that determinism is still guaranteed when a particular agent make anomalous memory accesses. The temporal dependencies are calculated on the state-transition diagram and the buffers sizing (*e.g.* for temporal variables and messages queues) are automatically performed too. It also performs graph generation for system safety analysis and check and sizing checks.

At present, the kernel is already available for Motorola m68k target systems, and studies and development began for Intel IA32 targets. Two industrial transfers are already under way.

Safety Oriented Embedded runtime

In order to achieve run time safety, it is a necessity to use all possible available mechanisms in a computer system. This has been done in the OASIS system. Indeed, privileged mode of processors limited only to critical run-time sections and memory protection between agents is achieved through MMU. At run-time, compilation deduced criterion like correct processing chaining and commutation or deadline checks are performed by the OASIS runtime environment without time penalty, thus enabling to accurately ensure that all agents behaviors are nominal and that any faulting task can not have any incidence on the other. From design to execution of the application, all possible mechanism allowing an early fault detection are implemented so that the deterministic behavior of an OASIS application is also ensured when anomalous behavior appears within agents. It also ensures data coherency and copies messages or data flows between agents. These are the specific tasks of the micro-kernel. This micro-kernel, whose specifications shall be shown in the full paper, is not a classical real-time monitor but simply execute and check control graphs. This micro-kernel is dedicated to this only task and is application independent and so is its validation. It is also aimed to be as portable as can be (mostly programmed in ANSI-C and as few assembly as possible).

The modularized approach of OASIS applications and their determinism in execution (so that the same causes induce the same effects) enables both system safety thanks to tasks confining, and a possible incremental per agent based design of the system, thus allowing easier and quicker design, implementation and checking phases.

Conclusion

This approach ensures strong safety properties like data coherency, a real-time unvarying and deterministic behavior, as well as fault detection and confining mechanisms. Determinism is a master key and a necessity in order to perform relevant tests. OASIS enables to reach such an objective with the further bonus of an easier design, implementation, testing phases and validation of a real-time system. It solves lots of classical issues encountered in real-time system design and programming. It implements a number of innovating features making a quantum leap in real-time methods. All tools are already fully operational and the project has a great vitality thanks to a number of related work, in order to take advantage of distributed systems, their synchronization or in another scope to compute tight evaluations of Worst Case Execution Time with cache memories on a multitasking system. As a new OS approach for embedded real-time applications, it solves a number of issues of primary importance for highly safety sensitive applications, like I&E/A systems in nuclear power plants which was the OASIS original target. Nonetheless, its intrinsic qualities make it become a sensible choice also for mainstream safety oriented applications.

References

- [Bre01] E. Bretz. By-wire cars turn the corner. *IEEE Spectrum*, pages 68–73, April 2001.
- [DAD00] Vincent David, Christophe Aussaguès, and Jean Delcoigne. Seeking a deterministic multitask framework for safety critical systems: the oasis approach. In *ANS/ENS, International Topical Meeting on Nuclear Plant Instrumentation*, 2000.
- [Kop] H. Kopetz. The time-triggered approach to real-time system design. In SpringerVerlang, editor, *Predictably Dependable Computing Systems, ESPRIT basic research series*.

Operating System Support for Massive Replication

Arun Venkataramani Ravi Kokku Mike Dahlin

{*arun,rkoku,dahlin*}@cs.utexas.edu

Computer Sciences, The University of Texas at Austin, USA

1 Introduction

The increasing number of devices used by each user to access data and services and the increasing importance of the data and services available electronically both favor “access-anywhere” network-delivered services. Unfortunately, making such services highly available is difficult. For example, even though end servers or service hosting sites advertise an availability of “four nines” (99.99%) or “five nines” (99.999%), the end-to-end service availability (as perceived by clients) is typically limited to two nines because of poor wide area network availability [6]. Moreover, although network bandwidths are improving quickly, network latencies are much more difficult to improve in wide area networks, which limits performance for access-anywhere services if those services are delivered from a single location.

This paper first argues that operating systems should provide support for massive replication of data and services. In particular, we argue that (1) technology trends favor “wasting” surprisingly large amounts of bandwidth and storage in order to improve availability or latency and (2) system support for massive replication is needed to realize these benefits because hand-tuning by engineers will not work.

This paper then outlines areas where operating system support can facilitate massive replication. We conclude that although a number of useful building blocks exist – particularly in the area of end-host resource management – additional work is needed to develop scalable end-to-end network support for massive replication, to develop client or edge-server support for simultaneously hosting large numbers of applications with essentially unlimited resource demands, and for developing end-to-end abstractions that make programming massive replication applications simple.

This work was supported in part by Tivoli Software, IBM Software Group and the Texas Advanced Technology Program through Faculty Partnership Awards. Dahlin was also supported by an NSF CAREER award (CCR-9733842) and an Alfred P. Sloan Fellowship.

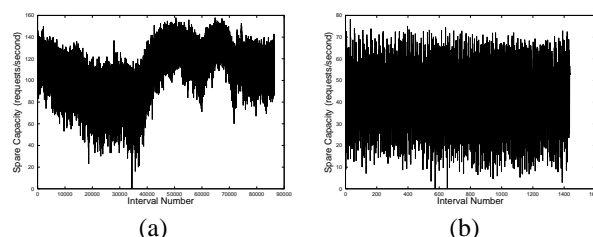


Figure 1. Server loads averaged over (a) 1-second and (b) 1-minute time scales for the IBM sporting event workload.

2 Case for Operating System Support

Operating system support for massive replication is motivated by two factors. First, technology trends suggest that massive replication will be an important building block for a wide range of services. Second, these same trends suggest that it will be difficult to successfully hand-tune applications that use massive replication.

Moving and storing electrons is extremely cheap, so it can make sense to “waste” many electrons to improve human-perceived availability and latency. In particular, the rapidly falling cost of network bandwidth [5, 14] and disk storage [8] – each improving at nearly 100% per year – may call for more aggressive replication than intuition might first suggest. Gray and Shenoy [10] describe a back of the envelope analysis that compares the dollar value of caching data versus the dollar cost of waiting while the data are refetched at some time in the future [10]; Chandra et. al [6] extend this model to consider prefetching and conclude that using an assumption similar to Gray and Shenoy’s estimates of network and disk costs in the year 2000, a system may be economically justified in prefetching an object even if there is only a 1% chance that that object will ever be used. In his Master’s thesis [5], Chandra argues that even more aggressive replication could be justified in the future given disk and network cost trends and given the desire to improve end-to-end availability not just performance.

Care should be taken in interpreting these results. If everyone started prefetching this aggressively tomorrow, the Internet would likely be overwhelmed. One way of viewing this calculation is that it suggests that economic incentives may exist to grow network capacity over time to accommodate increasingly aggressive prefetching.

A second technology factor that favors massive replication is the burstiness of demand workloads. Figure 1, from Chandra's thesis, shows the request load on an IBM server hosting a major sporting event during 1998 averaged over 1-second and 1-minute intervals. Server loads are bursty at many time scales with significant differences between peak and trough loads. Similar patterns have been noted elsewhere [7]. This burstiness suggests that systems are likely to be built with considerable spare capacity in order to accommodate bursts of load; this spare capacity can often be used to support aggressive replication. Conversely, systems built with the capacity to support aggressive replication will also benefit from an increased ability to handle large bursts of load.

Given these technology and workload trends, it seems likely that large numbers of applications will seek to make use of aggressive replication to improve availability or performance or both. For example, content distribution networks may wish to send updates to replicas before the new versions of objects are requested by clients [20]; multi-replica file systems may wish to immediately propagate all updates to maximize availability and consistency [21]; peer-to-peer systems may wish to replicate directory information [17] or data [20] to multiple replicas; systems supporting mobile clients may wish to replicate portions of file systems or databases to many client machines for disconnected access [12, 16]; and WAN enterprise servers or third-party file service providers [2] may wish to replicate the contents of file systems to geographically-distributed installations to provide the availability and responsiveness of local file systems while providing the global consistency of a WAN file system.

Although the opportunity for aggressive replication exists, it will be difficult for applications to take advantage of massive replication without end-to-end system support. A well-known problem with prefetching is that it consumes more resources than demand replication because some prefetched data is not used. Both *self-interference* – a prefetching application should ensure that its prefetch requests do not interfere with its demand requests – and *cross-interference* – prefetch requests should not interfere with other applications' demand requests – should be minimized.

Most often, this problem is addressed by hand tuning. For example, many prefetching algorithms estimate the probability that an object will be used, and then prefetch objects whose probability of use exceeds a threshold [9, 20].

Unfortunately, hand tuning seems unlikely to work for WAN massive replication.

- First, as noted above, intuition may be a poor guide for balancing the benefits and costs of prefetching. For example, Duchamp [9] selects a prefetch threshold of 25% as a reasonable balance between wasted bandwidth and latency reduction; the analysis discussed above suggests that this threshold may be far too conservative in many environments.
- Second, because of network cost and disk storage cost technology trends, the break-even point for prefetching will change significantly from year to year.
- Third, because of bursty workloads, the spare capacity available at a given point in time will change from second to second or minute to minute.
- Fourth, the complexity of hand tuning, and the lack of end-to-end support to ensure that prefetching requests do not interfere with demand requests discourages deployment of aggressive replication applications by (a) making the implementation of such systems more complex or fragile or both and (b) forcing conscientious application writers to be extremely cautious in their designs.

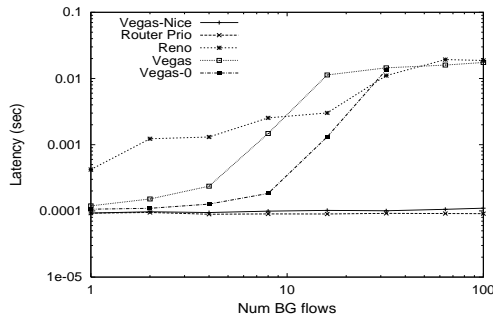
We believe that system support for massive replication should encourage development and deployment of applications that use massive replication to improve availability and performance. In particular, such support would allow simple applications that just state what they wish to replicate; the underlying system should be “self-tuning” and replicate as much as can be done without interfering with demand requests.

3 OS Support for Massive Replication

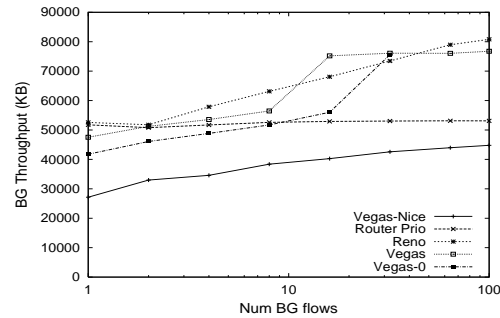
In this section we introduce a technique that allows us to allocate network resources in a manner such that a flow can be transmitted across the network without interfering with the flows already present. Thus, if there is spare capacity in the network, such flows can be efficiently transmitted; if not, the network will automatically ensure minimal self and cross-interference with other flows. We then discuss resource management issues at the end-stations to minimize interference between competing applications.

3.1 Network

Aggressively replicating and maintaining copies of data across the Internet can potentially consume virtually unbounded amounts of network bandwidth and interfere with existing applications. Since the network is a shared resource, it is essential to ensure minimal interference. However, a key question is whether network resource management be done in the network or at the end? One one hand



(a) Foreground latency vs. number of background flows



(b) Background throughput vs. number of background flows

Figure 2. Performance comparison of Reno, Router Prioritization, Nice and Vegas

simple prioritization schemes implemented at the routers such as the ones proposed for DiffServ [1] can easily prevent low priority flows from interfering with high priority flows. Unfortunately, there are practical hurdles to the immediate widespread deployability of such schemes.

We examine the other approach to develop an end-to-end transport protocol that approximates router prioritization without actually modifying the routers. We have developed a new congestion control algorithm, namely TCP *Nice* [19] as an end-to-end solution to the problem of minimizing interference. Nice is a simple extension to the TCP Vegas [4] congestion control protocol. Vegas uses round trip time (RTT) to limit the number of packets enqueued by a flow in the bottleneck router, so it provides a reasonable basis for conservative end-to-end congestion control. Unfortunately, Vegas was designed to compete fairly with TCP-Reno, so using it does not prevent background flows from interfering with foreground flows. The Nice extension makes the protocol much more responsive to competing traffic by adding an additional RTT-based congestion detection rule and by backing off multiplicatively when this rule detects congestion.

It is simple to make a protocol that behaves less aggressively than Vegas. The challenge is to meet two conflicting goals. First, background flows should not interfere with (e.g., increase the latency of) foreground flows. Second, demanding background flows should be able to consume a large fraction of the bandwidth not consumed by foreground flows. Our preliminary simulation based analysis with Nice suggests that it meets these goals effectively.

The set of graphs in Figure 2 show results of our simulation experiments over a dumbbell-shaped network topology with one bottleneck link connecting 20 clients and a server. The workload used for the simulations is a 15 minute trace of HTTP requests logged by Squid at UC Berkeley and a set of permanently backlogged background flows.

Figure 2(a) plots the latency of foreground requests as

a function of the number of background flows for a network utilization of 50%, when the background flows use Reno, RouterPrio, Nice and Vegas respectively. It can be seen that while Reno causes the latency to blow up by an order of magnitude because of interference, Nice causes little increase in latency compared to router prioritization and increases gracefully with the number of background flows. Figure 2(b) shows that the throughput attained by the background flows gets reduced (almost halved when there is just one background flow), but is comparable with RouterPrio as the number of flows increases. However, the lower background throughput is a reasonable trade-off for the agility that Nice provides in avoiding interference. Reno and Vegas on the other hand steal bandwidth from the foreground requests and hence delay them considerably.

In order for the application to be able to specify whether the data being sent is background or foreground traffic, we propose to have the transport layer at the sender expose a suitable API. This functionality can be provided by a lightweight *interference manager* just above the transport layer. The interference manager decides whether to transmit the data as regular foreground traffic or use Nice with an appropriately tuned *niceness* level. The interface manager may also aggregate appropriate sets of background requests into a single flow so as to create constantly backlogged flows.

3.2 End-stations

Two issues complicate dealing with interference at end-stations – i) end systems should deal with efficient allocation of multiple resources like CPU, disk and memory. ii) deployment of large number of services together with aggressive replication by each of them effectively implies near infinite demand for resources. For example, CDNs have to deal with thousands of demanding services and mobile clients that are capable of disconnected services [6], have to deal with tens or hundreds of services all contending for the system resources.

With respect to the first problem, many existing tech-

niques can be adopted for efficiently allocating multiple resources such that interference is minimized. Resource containers [3] allow us to define resource principals appropriate to the application. Systems such as Qlinux [15] enable the provision of QoS guarantees with respect to CPU, disk and network bandwidth.

The second problem, resource management across multiple services involved in massive replication, is more challenging. These challenges will affect the design of operating systems for hosting systems and operating systems for light-weight clients. The Active Names system has been adapted to provide fair self-tuning division of resources across downloaded extension code modules [6]. These techniques could be adapted to partition resources for downloaded code at clients, proxies, and hosting centers. Hosting systems like Denali [11] focus on providing services to safely execute many services on a single physical machine. However, in order to support massive replication, Denali should also provide scalable multi-resource management that prevents demanding applications from interfering with each other.

4 Policy

Designing massive replication systems also throws up questions of policy, namely *object selection* - what to replicate and *placement* - where to replicate. Our past research shows that by prefetching objects based on their popularity and lifetimes [20], significant improvements in hit rates may be obtained. In [13] and [18], we present algorithms to place objects in a distributed caching system so as to maximize hit rates in scenarios constrained by space and bandwidth respectively.

In future, we intend to work on developing algorithms to place objects in more dynamic scenarios where both space and bandwidth are simultaneous constraints. These algorithms require gathering good statistics of object usage patterns. Statistics should be gathered both on global scale (across various services to reflect the effects of services on each other) and on local scale (within a service to measure the service access patterns). Gathering good statistics forces us to strike a tradeoff between the precision of measurements and the scalability of the gathering mechanism. We intend to quantify this tradeoff and study its effects.

5 Conclusion

In this paper we have presented the challenges involved in providing system support for massive replication and an overview of their effect on operating system design at the network, server and edge nodes. We have also presented TCP Nice, an end-to-end congestion control algorithm optimized to support background transfers. We argue that the

design of applications relying on replication can be made simpler if the underlying operating system provides self-tuning support for resource management.

References

- [1] <http://www.ietf.org/html.charters/diffserv-charter.html>.
- [2] Akamai. Fast internet content delivery with freeflow. In *White Paper*, Nov 1999.
- [3] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, 1999.
- [4] L. Brakmo, S. O'Malley, and L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of SIGCOMM'94 Conference*.
- [5] B. Chandra. Web workloads influencing disconnected services access. Master's thesis, UT Austin, 2001.
- [6] B. Chandra, M. Dahlin, L. Gao, A. Khoja, A. Razzaq, and A. Sewani. Resource management for scalable disconnected access to web services. In *WWW10*, May 2001.
- [7] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centres. In *SOSP 2001*.
- [8] M. Dahlin. Historical disk storage costs, mar 2002. <http://cs.utexas.edu/~dahlin/techTrends/data/diskPrices/data>.
- [9] D. Duchamp. Prefetching Hyperlinks. In *Proceedings of the USITS*, October 1999.
- [10] J. Gray and P. Shenoy. Rules of Thumb in Data Engineering. In *"Proc. 16th Internat. Conference on Data Engineering"*, pages 3–12, 2000.
- [11] S. Gribble, A. Whittaker, and M. Shaw. Denali: Lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, Univ. of Washington, 2002.
- [12] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [13] Madhukar Korupolu and Mike Dahlin. Coordinated placement and replacement for large-scale distributed caches. In *Workshop On Internet Applications*, June 1999.
- [14] A. Odlyzko. Internet growth: Myth and reality, use and abuse. *Journal of Computer Resource Management*, 2001.
- [15] V. Sundaram, A. Chandra, P. Goyal, P.J. Shenoy, J. Sahni, and H.M. Vin. Application performance in the QLinux multimedia operating system. In *ACM Multimedia*, 2000.
- [16] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of SOSP 1995*, pages 172–183, December 1995.
- [17] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proceedings of ICDCS 1999*, May 1999.
- [18] A. Venkataramani, M. Dahlin, and P. Weidmann. Bandwidth constrained placement in a WAN. In *PODC*, Aug 2001.
- [19] A. Venkataramani, R. Kokku, and M. Dahlin. System support for background replication. Technical Report TR-02-30, UT, Austin Department of Computer Sciences, May 2002.
- [20] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. Potential costs and benefits of long-term prefetching for content-distribution. In *WCW*, June 2001.
- [21] H. Yu and Amin Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proceedings of SOSP 2001*, 2001.

Pangaea: a symbiotic wide-area file system

Yasushi Saito and Christos Karamanolis
 HP Labs, Storage Systems Department
 1501 Page Mill Rd, Palo Alto, CA, USA.
 {ysaito,christos}@hpl.hp.com

1 Introduction

Pangaea is a planetary-scale file system designed for large, multi-national corporations or groups of collaborating users spread over the world. Its goal is to handle people's daily storage needs—e.g., document sharing, software development, and data crunching—that can be write intensive. Pangaea uses *pervasive replication* to achieve low access latency and high availability. It creates replicas dynamically whenever and wherever requested, and builds a random graph of replicas for each file to propagate updates efficiently. It uses an optimistic consistency semantics by default, but it also offers a manual mechanism for enforcing consistency. This paper overviews Pangaea's philosophy and architecture for accommodating such environments and describes randomized protocols for managing large numbers of replicas efficiently.

Pangaea is built by federating a large number of unreliable, widely distributed commodity computers, provided by the system's users. Thus, the system faces continuous reconfiguration, with users moving, companies restructuring, and computers being added and removed. In this context, the design of Pangaea must meet three key goals:

Performance: Hide the wide-area networking latency; file access latency should resemble that of a local file system.

Availability: Not depend on the availability of any specific (central) servers; Pangaea must *adapt* automatically to server additions and failures without disturbing users.

Autonomy: Avoid centralized management; each node in the system should be able to perform its own resource management, e.g., the amount of disk space to offer and when to reclaim it.

To achieve these goals, we advocate a *symbiotic* approach to the system design. Each server should be able to function autonomously, and serve to its users most of their files even when disconnected. However, as more computers become available, they should dynamically adapt and collaborate with each other in a way that enhances the overall performance and availability of the system. Pangaea is an example of this approach.

1.1 Pangaea Overview

Pangaea builds a unified file system over thousands of servers (nodes) distributed over the world. We currently assume that these servers are trusted. Pangaea addresses the aforementioned goals via *pervasive replication*; it creates replicas of a file or a directory dynamically whenever and wherever requested and distributes updates to these replicas efficiently. Thus, there may be billions of files, each replicated on hundreds on nodes. Files shared by many people (e.g., the root directory) are replicated on virtually every node, whereas users' personal files reside on their local nodes and only a few remote nodes for availability. Pangaea demands no synchronous coordination among nodes to update file contents or to add or remove replicas. All changes to the system are propagated epidemically in the background.

Pervasive replication offers three main advantages: 1) provides fault tolerance, stronger for popular files; 2) hides network latency; 3) supports disconnected operations by containing a user's working set in a single server. These are key features for realizing the symbiotic approach in Pangaea's design.

1.2 Related work

The idea of pervasive replication is similar to persistent caching used in systems such as AFS, Coda [4] and LBFS [5]. However, Pangaea offers several additional advantages. First, it provides better availability. When a node crashes, there are always other nodes providing access to the files it hosted. Secondly, the decentralized nature of Pangaea also allows for better site autonomy; it lets any node be removed or replaced at any time transparently to the user. Finally, Pangaea achieves better performance by creating new replicas from a nearby existing replica and by propagating updates along fast network links. In this sense, Pangaea provides a generalization of the idea of Fluid replication [2] that utilizes surrogate Coda servers placed in strategic (but fixed) locations to improve the performance and availability of the system.

Mobile data sharing services, such as Lotus Notes [3], Roam [7], and Bayou [6], allow mobile users to replicate data dynamically and work disconnected. However, they lack a

replica location service. Humans are responsible for synchronizing devices to keep replicas consistent. In contrast, Pangaea keeps track of the location of replicas and distributes updates proactively and transparently to the users.

Farsite [1] is similar to Pangaea in that it provides a unified file system over pervasive number of nodes. The two systems, however, have different focuses. Farsite builds a reliable service on top of untrusted desktop nodes by using Byzantine consensus protocols, but it is not concerned with reducing latency. On the other hand, Pangaea assumes trusted servers, but it dynamically replicates files closer to their point of accesses to minimize the use of wide-area networks.

Recent peer-to-peer data sharing systems, such as CFS, Oceanstore, and PAST, build flat distributed tables using randomization techniques. Although Pangaea shares many of their goals—decentralization, availability and autonomy—its applications are different. In Pangaea, replicas are placed by user activity, not by randomization; files encounter frequent updates and are structured hierarchically. These differences force Pangaea to maintain a graph of replicas explicitly.

1.3 Challenges in Pangaea

While offering many benefits, pervasive replication introduces fundamental challenges as well. The first is the computational and storage overhead of meta-data management. While this problem is genuine, servers in cooperative-work environments, which we target initially, are known to waste much of their resources anyway [1].

The second challenge is to design algorithms for keeping track of a large number of files and replicas in a decentralized and highly available way. To address this problem, Pangaea builds a random graph over the set of replicas of each file and uses the graph to locate replicas and distribute updates. We present simple randomized algorithms for maintaining these graphs in Section 2.

The third challenge is the difficulty of propagating updates reliably yet efficiently and providing strong consistency guarantees. We solve this challenge by the combination of two techniques: 1) dynamic overlaying of a spanning tree over the random graph, and 2) a mechanism to explicitly enforce replica consistency for demanding applications. We discuss these mechanisms in Section 3.

2 Managing replica membership

Pangaea experiences very frequent replica additions and removals, because it manages billions of files, each of which is replicated independently on many servers. Thus, it calls for available and cheap mechanisms to manage the replica membership efficiently for each file and distribute updates reliably among replicas. Pangaea decentralizes both the tasks to

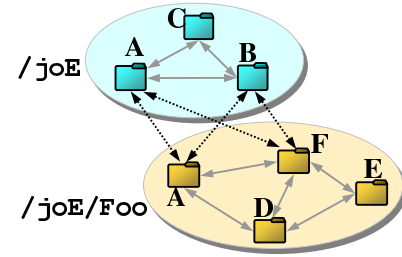


Figure 1: A replica graph. Each file (an oval) consists of its own set of replicas. Labels show the names of the servers that store replicas. Edges between replicas show acquaintance relationships. Updates to the file are propagated along these edges. Edges are also spanned between some of the file's replicas and those of its parent directory to allow for path traversal.

achieve this goal. There is no entity that centrally manages the replicas for a file. Instead, Pangaea lets each replica maintain links to k ($k = 3$ in our implementation) other random replicas of the same file. An update can be issued at any replica, and it is flooded along these edges (Section 3.1 describes a protocol for minimizing the flooding overhead.) Links to some of these replicas are also recorded in the entry of the parent directory, and they act as starting points file lookup (Section 2.3). Figure 1 shows an example. This design addresses our goals as follows:

Available update distribution: Pangaea can always distribute updates to all live replicas of a file even after k simultaneous server failures. With timely graph repairing (Section 2.2), it can tolerate arbitrary number of failures over time.

Available membership management: Pangaea lets a replica be added by connecting to any k live replicas, no matter how many other replicas are unavailable. This property essentially allows popular files to remain always available.

Available replica location: Each directory entry links to multiple replicas of the file. Directories themselves are replicated just like any regular file. Thus, Pangaea allows path name traversal even when some servers are unavailable.

Inexpensive membership management: Adding or removing a replica involves only a constant cost, regardless of the total number of replicas.

The following sections introduce randomized protocols for dynamically reconstructing a graph in response to replica addition or removal.

2.1 Adding a replica

When a file is first created, the user's local server selects a few (two in the current implementation) servers in addition to it-

self. We currently choose them randomly using a gossip-based membership service [9], but we plan to investigate more intelligent placement in the future. Pangaea then creates replicas of the new file on the chosen servers and spans graph edges among them. These replicas, called *golden replicas*, are registered in the parent directory and used for path traversals (Section 2.3).

A replica is added when a user tries to look up a file missing from her local server. First, the server retrieves the file's contents from a nearby replica (found in the parent directory) and serves them to the user immediately. In the background, the server adds k ($k = 3$ in our implementation) bi-directional edges to existing replicas. Parameter k trades off availability (tolerating a node or edge loss) and replica-management overhead. Lacking a central replica-management authority, Pangaea chooses peer replicas using *random walks*, starting from a golden replica and performing a series of remote procedure calls along graph edges. Our simulation shows that a random-walk distance of three is enough to keep the graph diameter at $O(\log N)$, or 5 for a 1000-replica graph.

Directories are files with special contents and are replicated using the same random-walk-based protocol. Thus, to replicate a file, its parent directory must also be replicated on the same server beforehand. This recursive step potentially continues all the way to the root directory. The locations of root-directory replicas are maintained using the gossip-based membership service.

2.2 Removing a replica

A replica is removed from a file's graph either involuntarily or voluntarily. It is removed involuntarily when its host server remains unavailable for a certain period (e.g., a week). When a replica detects the death of a graph neighbor, it autonomously initiates a random walk and spans an edge with another live replica. Starting the walk from a live golden replica ensures that the graph remains strongly connected.

A replica is removed voluntarily when the hosting server runs out of disk space or decides that the replica is not worth keeping. The protocol is the same as above, except that the retiring replica proactively sends notices to its graph neighbors, so that they can replace edges immediately to minimize the period of low graph connectivity.

2.3 Maintaining hierarchical name space using golden replicas

Pangaea's decentralized replica maintenance approach has some downsides. In particular, it is not trivial to ensure a minimum replication factor for a file, or to ensure that a file's replicas are reachable from the parent directory even after repeated replica additions and removals.

We solve these problems by marking some replicas "golden" (currently, replicas created initially are marked golden.) The golden replicas form a complete subgraph in the replica graph to let them monitor each other and maintain a minimum replication factor. Non-golden replicas store one-way edges to golden replicas and use them as starting points for random walks. The file's entry in the parent directory also points to the golden replicas. Otherwise, golden replicas act exactly like other replicas. This design allows non-golden replicas of a file to be added or removed without affecting its parent directory, and non-golden replicas of the directory to be added or removed without affecting children files.

Golden replicas must be given a high priority to stay on disk, since adding or removing a golden replica is a relatively expensive operation. For example, removing a golden replica involves picking one non-golden replica, "engolden" it, and telling all replicas of the file and the parent directory to update their pointers to the golden-replica set.

3 Propagating updates

A wide-area file system faces two inherently contradicting goals: providing high end-to-end availability and maintaining strong data consistency. We decided to favor availability when a trade-off is inevitable, based on a recent study that reveals that most instances of write sharing can be predicted easily, and that they demand consistency only within a window of minutes [8]. Thus, Pangaea maintains both replica membership and file contents completely *optimistically* without any synchronous coordination—any user can read or update any replica any time.

We discuss our solutions to three particular challenges that arise from our approach: efficient update propagation, providing strong consistency, and conflicting updates.

3.1 Efficient propagation using harbingers

Unlike the Web or p2p data sharing systems, file systems must handle updates efficiently since they are relatively common and synchronous, i.e., the user often waits for a write to complete before issuing subsequent operations. A naive (but safe) approach for distributing updates to replicas would be to "flood" the change along graph edges hop by hop. However, that would consume network bandwidth k times as much as optimal and increase propagation delay, especially for large updates. Thus, Pangaea uses a two-phase strategy to propagate updates that exceed a certain size (> 1 K bytes in the current implementation). In the first phase, a small message that only contains the timestamp of the update, called a *harbinger*, is pushed along the graph edges. When a node receives a new harbinger, it asks the sender to push the update body. When a node receives a duplicate harbinger without having received the actual update,

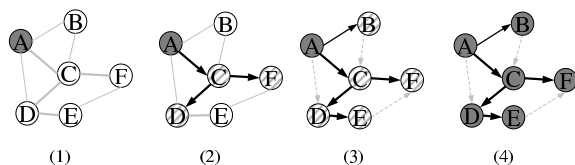


Figure 2: An example of update propagation in Pangaea. Thick edges represent fast links. (1) An update is issued at A. (2) A sends a harbinger to the fast edge, C. C forwards the harbinger to D and F quickly. (3) D forwards the harbinger to E. After some wait, A sends harbinger to B, and a spanning tree is formed. Links not in the tree are used as backups when some of the tree links fail. (4) The update's body is pushed along the tree edges. In practice, steps 3 and 4 proceed in parallel.

it asks the sender to retry later.¹ Moreover, before pushing (or forwarding) a harbinger to a graph edge, a node adds a slight delay inversely proportional to the estimated bandwidth of the edge. This way, for each update, Pangaea dynamically constructs a spanning tree whose shape closely matches the physical network topology. Figure 2 shows an example.

This harbinger algorithm yields two important benefits. First, it shrinks the effective window of replica inconsistency. When a user tries to read a file for which only a harbinger is received, she waits until the update body arrives. Because harbinger-propagation delay is independent of the actual update size, the chance of a user seeing stale file contents is greatly reduced. Second, it saves wide-area network usage, because it propagates updates along the fast edges. Our evaluation shows that this algorithm consumes only 5% more network bandwidth than an idealized optimal algorithm.

For users with strong consistency requirements, Pangaea also provides a mechanism to enforce consistency by synchronously circulating harbingers of recent updates. The user waits until all replicas of the files send back acknowledgments (or times out when remote nodes are unavailable). This mechanism may be initiated either manually by the user or automatically by Pangaea. The latter will be based on hints such as file names or application names, since write-sharing situations can often be predicted [8].

3.2 Conflict resolution

Because Pangaea requires no global coordination during any change, two nodes may issue different updates to the same object. Furthermore, some file operations—e.g., `mkdir` and `rename`—change two files (a file itself and its parent directory) that may encounter conflicts independently. We developed a proven-correct distributed protocol for this purpose. It is a variant of the version-vector-based algorithms used by Ficus

and Roam [7], but we omit the details due to space constraints.

4 Current status and future work

We are implementing Pangaea on Linux as a user-space loop-back NFS server. Preliminary evaluations indicate that, in a uniformly configured network, its performance is comparable to that of Coda. In a geographically distributed setting, Pangaea outperforms Coda by being able to transfer data from closer nodes, not just from a statically configured central server.

We identify two key areas of future research. First is the study of intelligent file placement heuristics. We plan to consider information such as storage capacity, network bandwidth, and content type to optimize both performance and availability. Second is security. We plan to study end-to-end data and metadata encryption and protocols for tolerating Byzantine servers.

References

- [1] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *ACM SIGMETRICS*, pages 34–43, June 2000.
- [2] Minkyong Kim, Landon P. Cox, and Brian D. Noble. Safety, visibility, and performance in a wide-area file system. In *FAST*. Usenix, January 2002.
- [3] C. Mohan. A database perspective on Lotus Domino/Notes. In *ACM SIGMOD*, page 507, May 1999.
- [4] Lily B. Mummert, Maria R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *SOSP*, pages 143–155, December 1995.
- [5] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *SOSP*, pages 174–187, October 2001.
- [6] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, pages 288–301, October 1997.
- [7] David H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, UC Los Angeles, 1998.
- [8] Susan Spence, Erik Riedel, and Magnus Karlsson. Adaptive consistency—patterns of sharing in a networked world. Technical report, HP Labs, 2002.
- [9] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *IFIP Int. Conf. on Dist. Sys. Platforms and Open Dist. Proc. (Middleware)*, 1998.

¹The sender must retry the propagation later because the node that send the harbinger the earliest may crash before sending the update body.

Replica Management Should Be A Game¹

Dennis Geels and John Kubitowicz
University of California
Berkeley, CA 94720 USA
{geels,kubitron}@cs.berkeley.edu

Abstract

We believe that large-scale replica management solutions should be based on an economic model. In this paper, we discuss the benefits provided by an economic approach and outline important directions for future research.

1. Introduction

As demand for information increases, centralized servers become a bottleneck. Content providers cope by distributing *replicas* of their files to servers scattered throughout the network. The replicas then respond to local client requests, reducing the load on the central server. *Replica Management* refers to the problem of deciding how many replicas of each file to distribute, and where to place them.

In a perfect system, replicas are placed near the clients that access them. Shrinking network distance decreases access latency and sensitivity to congestion and outages.

Also, exactly enough replicas should exist to handle the cumulative demand for each file. With too few replicas, servers become overloaded, and clients see reduced performance. Conversely, extra replicas waste bandwidth and storage that could be reassigned to other files, as well as the money spent to rent, power, and cool the host machine.

Replica management alternatives Several approaches to replica management have been developed. One solution, perhaps best embodied by Content Distribution Networks (CDNs)[1, 4, 20], involves deploying new machines throughout the network. These machines only host replicas of their company's content.

Peer-to-Peer storage systems, including FreeNet[6], Gnutella[2], and many research prototypes (e.g. Bayou[9] and CFS[8]), consist of independently owned and operated machines. Each machine controls its own set of replicas, but

freely stores and serves content produced elsewhere. Limited resources are usually handled with a simple cache algorithm such as LRU.

A third approach, sharing many P2P characteristics, applies concepts from Economics to the replica management problem[3, 12, 21, 23]. Here, machines earn (real or virtual) money by hosting replicas and use that money to purchase access to replicas hosted by other machines.

Replica management economies In these economic systems, individual machines are *autonomous*—free to choose which replicas they host. They may make such decisions using simple on-demand algorithms or complicated predictive methods. In fact, each could use a different algorithm. Together, payment-based cooperation and ambivalence towards local server algorithms are the defining characteristics of a *Replica Management Economy*, or *RME*.

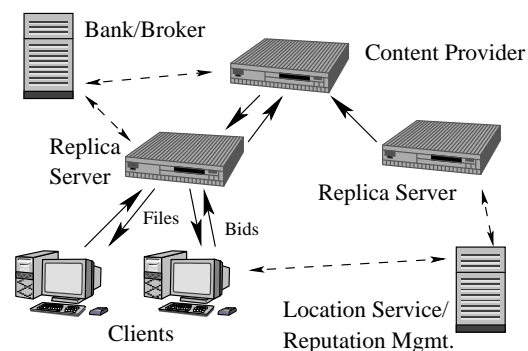


Figure 1. A simple RME. Clients bid for access to local replicas. Third party machines provide peripheral services, including currency exchange and reputation management.

In the following section we argue that the RME is a very flexible and robust solution to large and complex replica management problems. We then present a few examples of successful experiments with RMEs. Finally, we discuss current limitations and directions for future work.

¹This research supported by NSF career award #ANI-9985250 and NFS ITR award #CCR-0085899. Dennis Geels is supported by the Fannie and John Hertz Foundation

2. Why an economic model?

Automatic resource management When two clients compete for access to a server with limited resources, some decision must be made as to which requests are *more important*. An economic approach defines the *importance* of a request as the amount the requester is willing to pay. A client provides useful feedback about its priorities by offering to pay servers more for certain replicas. One could argue that money is really nothing more than society's best attempt at producing a ranking of the relative importance of most everything.

The economic model also helps a replica system cope with fluctuating demand. As hot spots appear, such as when important news breaks or a popular web site links to a normally-low-traffic page, the high demand increases the cost that servers can charge for access to replicas of the hot content. This increase encourages other servers to host a replica, distributing the load and sharing the profit. Similarly, an economy can adapt to the addition or deletion of machines without intervention from human administrators.

Also, an economy provides an easy way to decide *when* to add new servers to a system. System administrators, like capitalist entrepreneurs, can monitor price fluctuations for areas with consistently high prices, which suggests that client demand exceeds replica supply.

Scalability Replica Management Economies also share the scalability benefits of cooperative P2P alternatives. Their use of local, greedy control algorithms avoids the computation and bandwidth bottlenecks that may appear if storage allocation, network monitoring, and failure detection are performed by a central authority.

Guarantees through mechanism design One subfield of Game Theory, called Mechanism Design, studies techniques for setting system rules (algorithms, prices, etc.) in order to induce outcomes with certain desired properties. These properties may include cooperation, a balanced budget, and various definitions of "fairness".

As a simple example, we could define an economy in which clients and servers interact using a Second-Price Auction. Each client submits a bid for replica access; the server then awards access to the highest bidder but charges the amount bid by the runner-up. It can be shown[25] that this method guarantees that "rational" clients will bid honestly. Many generalizations of this simple second-price auction have been proposed which may prove useful in replica management economies.

Benefits in a federated environment A network of machines is said to be *federated* if the machines operate in separate administrative domains. They may cooperate to attain a common good, but each is autonomous and primarily concerned with its own success and profitability.

RMEs fit naturally in this type of environment, which motivates most of Microeconomics and Game Theory. RMEs explicitly deal with real trust and administrative boundaries, as well as real money. They assume that machines may often reject requests, will not always volunteer truthful information, and demand payment proportionate to the work they expend. These concepts usually must be grafted onto other systems before they can be deployed in a federated environment.

Benefits in a trusted infrastructure On the opposite end of the spectrum, one could imagine an environment containing a single administrative domain. All machines cooperate fully, accepting external storage and retrieval requests for the common good.

Despite their apparent differences, both content distribution networks and pure, cooperative P2P systems assume this environment. The former tend to employ a more global allocation algorithm and possibly restrict the set of machines that initiate the storage requests, but both approaches rely on the same inter-machine cooperation.

In contrast, machines in a replica management economy accept external requests only when paid enough to make the action worthwhile. There is no need in this environment for machines to maintain individual profitability; however, this restriction on cooperation can improve system robustness.

Unbounded cooperation, although conceptually simple and morally pleasing, allows a single machine to reduce the availability of many others. Poorly configured or broken machines may accidentally flood the system with unnecessary storage requests. Compromised machines may launch Denial of Service attacks. Or, perhaps more likely, greedy users will consume more resources than they should[14].

In an RME, faulty or malicious machines must pay for service, and their funds are finite. Overloaded machines can raise their prices until demand drops or the failed machines run out of money. Thus, unlike more trusting models, an RME bounds the impact of failure or active attack.

One could impose a similar bound on any replica management system; however, fixed bounds can be overly restrictive. They limit the flexibility of machines that are functioning perfectly yet require a great deal of resources. In an RME, the limit is soft; a machine can always acquire access to a replica if it is willing and able to pay enough. In Game Theory, this property is called *consumer sovereignty*.

Benefits in the internet The internet is arguably the most important environment to consider when designing a large-scale replica system. Like many networks, it is neither fully cooperative nor fully federated; it contains many competitive domains, each containing machines that cooperate more or less completely.

One could treat domains as opaque units and only impose a replica management economy among them. This ap-

proach would allow competitors to share resources safely.

One could also expose the machines in each domain and extend the economy to handle intra-domain interactions as well. As shown above, the economic model provides interesting benefits even within trusted domains.

Machines could still be programmed to favor others from their own domain. The RME does not prevent such *coalitional* activity; however, increasing the dependencies between machines decreases the robustness benefits of an RME. As in the real world, tying a greater portion of one's income or output to a favored trading partner or single resource is often risky. The lessons from Economics must be considered when programming members of an RME.

3. Previous results

A small number of previous projects have explored an economic approach to replica management. We summarize their results here, in order to frame the following discussion of future work.

Kurose and Simha [17] used an analytical model to examine the convergence rates of a decentralized allocation algorithm. They assumed a mostly-cooperative environment wherein machines redistributed files to needier machines.

Ferguson ([11] and later in [12]) implemented and measured the performance of a competitive bid-auction mechanism. He found that simple bidding mechanisms produced good allocations for a various access patterns.

Later, the Mariposa project [23] published a design for the most complete replica management economy to date. They implemented an auction mechanism that distributed database tables and queries among autonomous replicas. They found that their economic system balanced query load across replicas better than a static query optimizer.

Clearwater's book [7] and a paper from Tucker and Berman [24] are useful sources for other, less related work. The latter includes a discussion of reasons for which the economic paradigm has not yet been widely adopted.

Recent work applying Game Theory to computer systems seems to focus on networking problems, such as sharing the cost of multicast [10, 15] and handling congestion in the internet [16]. These papers show intriguing adaption of economic theory but are not directly applicable to replica management.

4. Directions for future research

In this section we address the major obstacles to widespread adoption of the RME in popular replica management systems.

Player design Individual machines must be programmed to prioritize their requests and set prices. In general, a machine requires a *utility function*, which rates the relative

worth of sets of files. When deciding, for instance, whether to discard a replica to free resources, the machine simply compares the expected *utility* of each alternative.

The utility function could consider the storage and network resources consumed by the content it stores, the money it expects to receive in exchange for those resources, and the amount of money it plans to spend to acquire content in the future.

A simple utility function may assign a fixed worth to each request. We are currently investigating utility functions that favor clusters of files that are expected to be accessed in the near future.

The amount of computation and prediction required for a good utility function is an open problem. Evidence suggests that simple methods will do reasonably well [12].

Performance Greedy, decentralized algorithms rarely achieve the maximum level of performance achievable by centralized, analytical methods; however, one can sometimes bound the difference. The characteristics of economies and players that enable such bounds should be explored more fully.

Also, we should consider the total effect of an economic approach. One might prefer an RME, which requires extra machines and network resources, over an analytical model that requires more human intervention, is less flexible, or imposes heavy control overhead.

Complexity Some may argue that an RME is harder to understand, and hence to control and repair, than a more centralized approach. For large networks, however, the complexity of any system soon exceeds the limits of direct analysis. Instead, we rely on abstractions, summaries, and models of the system's behavior. The field of Economics has developed many useful models that we may apply to the behavior of an RME.

Not group strategyproof The term *strategyproof* from Game Theory refers to games whose rules discourage rational players from lying. A game is *group strategyproof* if a group of players cannot benefit even if the entire group cheats together.

This restriction is often desirable, but very hard to guarantee. Real economies are not group strategyproof; they often develop overseer organizations and anti-trust legislation to prevent certain behavior by coalitions of greedy players. A replica management economy may require similar solutions. Reputation management systems may also help limit the spread of destructive coalitions.

Perhaps future results in Mechanism Design will better characterize rules that induce group strategyproofness.

Starvation Purely economic systems present the danger that poorer clients might be unable to afford reasonable ser-

vice. This problem, like the previous one, may be dealt with either through Mechanism Design or external restrictions.

For example, one could dedicate a small set of servers to serve one's clients for free. The other servers, which operate in the RME, would provide higher QoS guarantees for those able to pay.

Need electronic currency An RME requires a secure, efficient payment mechanism if its digital money is tied to "real" money. A large system must handle millions of transactions per second, and latency and availability requirements rule out centralized systems.

No existing system meets all of our requirements, but several warrant further research. Digital cash systems [5, 13] provide secure, anonymous, offline payments, but require significant computational overhead. Probabilistic methods [18, 22] amortize communication with a central bank across many transactions. Millicent [19] used symmetric-key cryptography to optimize the transaction phase, which required relaxing security goals.

5. Conclusion

We have argued for the Replica Management Economy as a robust, flexible solution for large-scale replica management. RMEs allow machines a level of autonomy that should be expected in a heterogeneous environment like the internet. They rely on an economic model of interaction that allows, yet flexibly bounds, cooperation across domains.

Much work remains in the design of RME protocols and local player algorithms. We are currently building an experimental testbed in which to explore this design space, within the framework of a large-scale storage system. We hope to develop a system that matches current methods in performance and surpasses them in robustness and flexibility.

References

- [1] Akamai technologies, inc. <http://www.akamai.com/>.
- [2] Gnutella. <http://www.gnutellanews.com/information/>.
- [3] Mojonation. <http://www.mojonation.net/>.
- [4] S. Acharya and S. B. Zdonik. An efficient scheme for dynamic data replication. Technical Report CS-93-43, Department of Computer Science, Brown University, 1993.
- [5] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. In *Communications of the ACM*, 1985.
- [6] I. Clark, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, Berkeley, CA, July 2000.
- [7] S. H. Clearwater, editor. *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific Press, 1996.
- [8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, October 2001.
- [9] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proc. of IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, 1994.
- [10] J. Feigenbaum, C. H. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. In *Proc. of ACM STOC*, 2000.
- [11] D. Ferguson. *The Application of Microeconomics to the Design of Resource Allocation and Control Algorithms in Distributed Systems*. PhD thesis, Columbia University, 1989.
- [12] D. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic models for allocating resources in computer systems. In S. H. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*. 1996.
- [13] N. Ferguson. Single term off-line coins. In *EUROCRYPT*, 1993.
- [14] G. Hardin. The tragedy of the commons. *Science*, 162:1243–1248, 1968.
- [15] K. Jain and V. Vazirani. Group strategyproofness and no subsidy via lp-duality. 1999.
- [16] P. Key and D. McAuley. Differential qos and pricing in networks: where flow-control meets game theory. In *IEE Proceedings Software*, 1999.
- [17] J. F. Kurose and R. Simha. A microeconomic approach to optimal resource allocation in distributed computer systems. *IEEE Transactions on Computers*, 8(5):705–717, May 1989.
- [18] R. Lipton and R. Ostrovsky. Micro-payments via efficient coin-flipping. In *Financial Cryptography Conference*, 1998.
- [19] M. Manasse. The millicent protocols for electronic commerce. In *USENIX Workshop of Electronic Commerce*, 1995.
- [20] M. Rabinovich and A. Aggarwal. Radar: A scalable architecture for a global web hosting service. In *The 8th Int. World Wide Web Conf*, May 1999.
- [21] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherpoon, and J. Kubiatowicz. Maintenance free global storage in oceanstore. In *Proc. of IEEE Internet Computing*. IEEE, Sept. 2001.
- [22] R. Rivest and A. Shamir. Payword and micromint: Two simple micropayment schemes. In *Security Protocols Workshop*, 1996.
- [23] J. Sidell, P. Aoki, S. Barr, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data replication in Mariposa. In *Proc. of IEEE ICDE*, pages 485–495, Feb. 1996.
- [24] P. Tucker and F. Berman. On market mechanisms as a software technique. Technical Report CMU-CS-87-143, U. C. San Diego, Dec. 1996.
- [25] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Finance*, 16:8–37, 1961.

Secure Coprocessor-based Intrusion Detection

Xiaolan Zhang Leendert van Doorn Trent Jaeger Ronald Perez Reiner Sailer

IBM T. J. Watson Research Center

Hawthorne, NY 10532 USA

Email: {cxzhang,leendert,jaegert,ronpz,sailer}@us.ibm.com

1 Introduction

The goal of an intrusion detection system (IDS) is to recognize attacks such that their exploitation can be prevented. Since computer systems are complex, there are a variety of places where detection is possible. For example, analysis of network traffic may indicate an attack in progress [11], a compromised daemon may be detected by its abnormal behavior [14, 12, 5, 10, 15], and subsequent attacks may be prevented by the detection of backdoors and stepping stones [16, 17].

The most popular architecture for IDSs is *host-based intrusion detection*, where the IDS runs as a monitor on its host and collects information used to identify possible intrusions on that host. Since the compromise of any system service generally results in the compromise of the operating system's trusted computing base (TCB), the IDS is also susceptible to compromise, and thus cannot be trusted.

In this paper, we examine the effectiveness of *secure-coprocessor-based intrusion detection*. In this case, the IDS is run on a coprocessor rather than on the host. Thus, a compromise of the host does not affect the coprocessor, and self-protection of the IDS monitor is achieved. Since a coprocessor can see the memory of the host, a coprocessor IDS can verify that the host's state is correct. However, a coprocessor IDS cannot interpose the host's execution the way that a host IDS can. Therefore, we need to identify a new approach that enables effective detection given the external nature of the coprocessor.

The remainder of the paper is structured as follows. In Section 2, we define coprocessor-based intrusion detection. In Section 3, we discuss a series of applications possible with this approach, and describe experiments that show the kinds of attacks that can be successfully detected by a coprocessor. In Section 4, we discuss limitations of this approach and how it can be extended to take preventive steps when anomalies are detected. Section 5 concludes.

2 Coprocessor-based Intrusion Detection

Coprocessor-based intrusion detection means that host data is collected and processed by software running on a coprocessor rather than the host itself [4]. Typically, a coprocessor shares interfaces with the host processor that enables it to examine and perhaps modify the state of the host. The number and choice of interfaces determines the degree to which intrusion detection is possible. We use a secure coprocessor because it offers additional security features that are desirable for an IDS (see Section 2.2).

2.1 Secure Coprocessors

A *secure coprocessor* is a tamper-resistant computing device designed to perform critical tasks in an environment in which physical attacks are possible. Such a device can be used to securely boot the host system into a known state.

In the context of this paper, we examine use of the IBM 4758 PCI Cryptographic Coprocessor [1, 7, 13]. The 4758 consists of a CPU, volatile and non-volatile memory, and cryptographic accelerators. It is wrapped inside a tamper-responding secure boundary. The device communicates with the host via the PCI bus, whereby it can issue commands to operate on system memory.

The software architecture of the IBM 4758 device is designed to support generic security applications [8]. The software insures that the device boots securely, and that only authorized programs can execute on the device. The device also comes with factory-installed certificates that allow it to authenticate itself to external entities.

2.2 Advantages

Compared to host-based intrusion detection, the use of a secure coprocessor for intrusion detection has the following advantages:

1. **Independence from the host OS.** The secure coprocessor is an autonomous subsystem that has its own

operating system and application software. Its tamper-resistance also provides strong integrity protection for the IDS.

2. **Narrow interface.** The interface used for communicating between the host and the secure coprocessor is simplistic and well-defined. It is therefore much more difficult to exploit the interface and launch attacks.
3. **Secure boot.** The secure coprocessor can be used to boot the host into a known state in which invariants can be defined.
4. **Trusted observer.** Since the secure coprocessor is designed to protect its authentication keys against almost any attack, any authenticated statements made by the secure coprocessor can be fully trusted. This is very useful in a scenario where multiple coprocessors collaborate on a task or the IDS data is consumed by a remote entity.

2.3 Monitoring

In addition to self-protection, an effective IDS must be able to monitor the controlled operations that may lead to an intrusion. In a host-based IDS, system calls and key kernel operations are often interposed such that IDS data can be collected and analyzed. In a coprocessor-based IDS, monitoring cannot be done by interposition. The host will continue to execute, so the IDS paradigm must be altered to suit this environment.

Instead of interposing operations, we propose that the coprocessor IDS base its analysis on system invariants. The host system as a whole maintains certain integrity properties (invariants) when it is functioning correctly (i.e. it has not yet been compromised). Since it can boot the host into a known state, the coprocessor IDS can be expected to know certain host OS state, such as the location and value of key data structures. Given key data structures and invariants on their values and the way in which values are allowed to change, the coprocessor can sample the host OS to verify that the invariants are still held.

3 Applications

In this section we discuss several monitoring applications that can be implemented on the secure coprocessor.

3.1 Checking Kernel Data Structures Invariants

The first set of invariants we will examine concern in-memory kernel data structures. We can view the OS as a state machine whose states are stored in a collection of internal data structures. Examples of such data structures include *task_struct*, the data structure that abstracts the notion

of process, and *inode*, the data structure for representing a file¹. The operating system reacts to external events (such as system calls or network packet arrivals) by performing appropriate modifications on these kernel data structures. Assuming that, when the system is in a secure state, the values of these kernel data structures are consistent and exhibit a set of invariants, but when the system is compromised these invariants no longer hold, the monitoring system can detect break-ins (or break-in attempts) by continuously checking for consistencies of crucial kernel data structures.

Our approach is to be distinguished from previous approaches [12, 14], which focus on the events that cause the kernel to enter an illegal state, rather than on the states themselves. For example, in Forrest's approach, one first profiles the target application and collects a database of legitimate system call sequences (signatures) made by the application. In the production system, the OS monitors system calls executed by these applications, and issues a warning if a sequence does not match any in the database. Since events are program dependent, e.g., different programs typically have different signatures, one thus needs to maintain an extensive database of signatures covering all programs. Frequent software upgrades further complicate this problem. Our approach, on the other hand, tries to abstract the validity of states into invariants. Because we use abstractions, we keep less information. In addition, the invariants describe properties of the kernel only, and thus are much more stable.

3.1.1 Determining Kernel Invariants

To find out what the invariants are, we implemented a kernel module that intercepts system calls and records the values of crucial data structures at the entry of each system call. We then compare the value trace of a correct OS with that of a compromised OS for a given attack taken from an database of known exploit programs, and search for systematic differences between the pair of values. The systematic differences potentially highlight the invariants that are violated.

We define two types of invariants: *global invariants* and *application-specific invariants*. Global invariants are invariants that apply across the entire operating system, independent of the programs running on top of the OS. Examples of global invariants include immutability of the kernel image and images of crucial system programs, and immutability of kernel data structures such as system call tables. Application-specific invariants, on the other hand, depend on the specific nature of the application represented by the kernel data structures in question. For example, a normal user program's *uid* should never change to root. This

¹Unless otherwise explicitly stated, we base our discussion on the Linux operating system.

invariant certainly does not apply to programs such as *su*.

3.1.2 Detecting Violations

Once the invariants are determined, it is relatively easy to detect violations against these invariants. For global invariants such as immutability of kernel image, the monitor computes a checksum over the image at (secure) boot time, and periodically recomputes the checksum and compares it with the stored value. For application-specific invariants, the monitor determines the type of application by its name (i.e. the command line field of the task structure), and loads appropriate invariants according to the application type. It then periodically samples the values of relevant data structures and checks the values against the invariants.

3.1.3 An Example

Let's look at an example invariant that we derive by running a local-root exploit program [2] and comparing the changes in the *task_struct* values between a successful attempt and those of a normal user program. The attack program, *ptrace24*, works by exploiting the race between *ptrace* and *execve* and injecting arbitrary code into a *setuid* program. Table 3.1.3 shows the fields of the *task_struct* data structure that exhibit different change patterns depending on whether the attack succeeds or not.

We derive the following invariants for normal user-programs from the above data.

1. *uid* should remain the same throughout execution.
2. *euid*, *suid* and *fsuid* should not be different from the original *uid* for an extended period of time.

To check the invariants, the monitor periodically scans the values of the relevant fields of the task structure for active processes, and validates the values against the invariants. Note that the monitor needs to store the old value of *uid* for each process.

This simple example illustrates that it is possible to infer invariants by profiling known exploits and use the invariants to detect ill-behaved processes.

To test the generality of this invariant, we examined another exploit program [3] that uses a different technique to gain control of the system. A bug in the *traceroute* program causes buffer overflow and allows arbitrary code to be executed on the stack. The invariants are essentially the same as the *ptrace* exploit. This is not particularly surprising because both exploits attack the system by becoming the root, which requires a change of the *uid* field to *root*. However, it demonstrates that the invariants are applicable to exploits of the same nature (in this case, local root exploit

through *setuid* programs), and thus only one set of invariants are needed for these exploits even though they differ dramatically in the methodology of attacking.

3.2 File Integrity Checking

Another important correctness property of a system is the integrity of system files on disk. The monitoring system can independently scan the disk, compute checksums of system files, and compares the results against those stored in a database. This is similar to the Tripwire [9] commercial product. The difference being that the monitoring system and the checksum database reside on the secure coprocessor, instead of on the host system, and are thus not vulnerable to attacks.

3.3 Virus Detection

The monitoring system can also scan the entire memory for known viruses. Again because the monitor resides on the coprocessor, it is much less intrusive than a tool like Norton Utilities.

4 Discussion

Since the monitoring system is based on sampling, there is no guarantee that the attack is detected in time. However, we can reduce this likelihood through control of the placement of samples, the number of samples, the sampling period, and the period distribution. Sample placement is driven by the number of attacks that can be detected and the accuracy of the detection. Obviously, a single point that detects all errors with perfect accuracy would be the best case. Since this is unlikely, we can increase the number of samples until we have sufficient coverage. However, the number of samples is limited by the computing speed of the coprocessor.

One way to reduce this cost is to identify dependencies between sampling points. Only when one sample is triggered are its dependents samples, and others are delayed or removed temporarily. Another way is to adaptively change the sampling frequency based on the actual state of the system. For instance, we could raise the sampling rate when suspicious events are detected, such as when a process is running with root or *setuid* privilege, and slow down the rate once suspicious events cease to exist. This way, the time window for an undetected attack is smaller at times of higher risk. Finally, we can vary the periodic distribution of the sampling to reduce its predictability.

Another limitation of our current monitor is that it is based on the PCI bus which provides only limited control over the host. Ideally, we would like to be able to use the host JTAG bus [6]. The JTAG bus is a hardware debug facility that can be used to control peripherals in the host. That

Field	Sample Sequence of A Successful Attack				Sample Sequence of A Normal Process			
	1	2	3	4	1	2	3	4
flags	64	0	256	256	64	0	0	0
uid	500	500	0	0	500	500	500	500
euid	500	0	0	0	500	0	500	500
suid	500	0	0	0	500	0	500	500
fsuid	500	0	0	0	500	0	500	500
cap_effective	0	x	x	x	0	x	0	0
cap_permitted	0	x	x	x	0	x	0	0
user	x	x	y	y	x	x	x	x

Table 1. Sampled values of fields of *task_struct* for a successful attack and a normal user process. Shown above are a sequence of 4 sample points taken at 4 different system call entry points. For simplicity reasons, only a subset of sample points are presented here. For the flags field, 64 means forked but not exec, 256 means used privileges. For the cap_effective, cap_permitted, and user fields, x means non-zero value, and y means a non-zero value other than x.

is, stop the CPU, inspect its state, and resume execution, or inspect/change the state of memory or any other controller attached to the bus. The JTAG approach can thus take preventative/remedy steps when an anomaly is detected. There is however a tradeoff between cost and effectiveness. JTAG is chip-dependent and thus much more expensive than the generic PCI-based solution. However, if the PCI approach is promising we may explore the JTAG approach so we can assert greater control over the host.

5 Conclusion

In this paper we proposed building intrusion detection systems using external secure coprocessors. Because the coprocessor runs independent of the host, a compromise of the host does not affect the functionality of the IDS. The additional security features of the coprocessor ensure that the host starts from a secure state, and that messages sent by the coprocessor can be authenticated and trusted. We discussed a series of possible monitoring applications, and our early results demonstrated the viability of this approach.

References

- [1] IBM PCI Cryptographic Coprocessor General Information Manual, May 2002. Available at <http://www.ibm.com/security/cryptocards>.
- [2] Ptrace2.4. Available at <http://packetstormsecurity.org/0203-exploits/ptrace-dark.c>.
- [3] Traceroute exploit + story. Available at <http://security-archive.merton.ox.ac.uk/bugtraq-200010/0084.html>.
- [4] J. M. A. Mishra and W. Arbaugh. The coprocessor as an independent auditor. Available at <http://www.missl.cs.umd.edu/komoku/documents/coauditor.ps>.
- [5] S. N. Chari and P. Cheng. Bluebox: A policy driven, host-based intrusion detection system. In *Proceedings of the 2002 Network and Distributed System Security*, February 2002.
- [6] IEEE. IEEE standard test access port and boundary-scan architecture, IEEE std 1149.1b-1994.
- [7] R. P. R. S. L. v. D. S. W. S. J. Dyer, M. Lindemann and S. Weingart. Building the ibm 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, 2001.
- [8] S. W. S. J. Dyer, R. Perez and M. Lindemann. Application support architecture for a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference (NISSC)*, October 1999.
- [9] G. H. Kim and E. H. Spafford. Experiences with tripwire: Using integrity checkers for intrusion detection. In *System Administration, Networking and Security Conference III*, 1994.
- [10] E. G. M. Bernaschi and L. V. Mancini. Operating system enhancements to prevent the misuse of system calls. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 174–183, 2000.
- [11] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.
- [12] A. S. S. Forrest, S. Hofmeyr and T. Longstaff. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, 1996.
- [13] S. W. S. Smith, R. Perez and V. Austel. Validating a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference (NISSC)*, October 1999.
- [14] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.
- [15] D. Zamboni. Using internal sensors for computer intrusion detection, 2001. CERIAS Technical Report 2001-42, CERIAS, Purdue University.
- [16] Y. Zhang and V. Paxson. Detecting backdoors. In *Proceedings of 9th USENIX Security Symposium*, August 2000.
- [17] Y. Zhang and V. Paxson. Detecting stepping stones. In *Proceedings of 9th USENIX Security Symposium*, August 2000.

THINK: A Secure Distributed Systems Architecture

Christophe Rippert Jean-Bernard Stefani
 LSR-IMAG laboratory, SARDES project, CNRS-INPG-INRIA-UJF
 {Christophe.Rippert, Jean-Bernard.Stefani}@inria.fr

1 Introduction

In this paper, we present THINK, our distributed systems architecture, and the research we have conducted to provide the system programmer with an architecture he can use to build efficient and secure operating systems. By specifying and implementing elementary tools that can be used by the system programmer to implement a chosen security policy, we prove that flexibility can be guaranteed in an operating system without compromising security. Our work focuses on protection against denial of service attacks which compromise the system fairness in resource multiplexing and can cause the system to stall due to resource starvation.

We first briefly describing the THINK architecture before positioning our contribution against related work. We then present the elementary tools we have specified to ensure quality of service in THINK, before detailing the software memory isolation tool we have implemented and tested. We conclude by a concrete example of the utilisation of these tools.

2 The THINK architecture

2.1 Presentation

The distributed systems architecture THINK is a platform for the development of distributed operating systems kernels. The goal of the THINK architecture is to ease the development of efficient, flexible, and secure operating systems. THINK provides the system programmer with interfaces that reify the underlying hardware, and optional system abstractions proposed as libraries. The development of a kernel with THINK is made easier by its object-oriented framework, since in THINK, all resources (both hardware and software) are considered as objects. These objects export interfaces which define their behaviour and make them accessible to other objects. Each interface has a name in a given naming context, and is linked to other interfaces by bindings. A binding is essentially a communication channel between two objects. These bindings can take many forms, as simple as the association between a variable name

and its value, or more complex like a binding over a network between two objects on different machines. Bindings are created by dedicated objects called binding factories, those main function (i.e. creating bindings) can be freely extended to enforce a chosen behaviour. Finally, objects can be grouped in domains according to a common property (e.g. security domains, fault-domains, etc). A more detailed presentation of THINK can be found in [1].

2.2 Related work

Compared to the OSKit [2] framework and set of libraries, THINK provides a better flexibility since all components are independent. The KaffeOS [3] project proposes some techniques to preserve quality of service in a Java environment, using the type safety properties of the language. Similarly, the SPIN [4] extensible operating system uses the properties of the Modula-3 language to permit the safe binding of modules in the operating system. In THINK, we aim to remain independent of the component development language. In the DTOS [5] project, policy-neutral security is enforced by way of security servers, which check that inter-component calls are allowed. This requires a modification of the component code, whereas in THINK, binding factories can make the security checks, which ensures a complete independence of the component code. The Scout/Escort [6] project focuses on protection against denial of service attacks by defining the I/O path abstraction. However, this does not take into account the resources allocated in the operating system kernel, whereas in THINK, we aim toward a global view of the resources which enable the system to associate each allocated resource with the benefiting user. This point of view is close to the resource containers abstraction [7], although this work has been conducted in monolithic kernels whereas in THINK we advocate a more modular architecture. The exo-kernel [8] advocates the same philosophy of a minimalist kernel, although it does not propose an object framework for the components as in THINK. Finally, the memory isolation tool presented below is inspired from software fault isolation [9], but in THINK we are able to isolate a single process whereas soft-

ware fault isolation works with binary object files representing a whole module.

3 Preserving quality of service

The THINK architecture offers an unparalleled flexibility for the construction of operating system kernels, since it allows access to the hardware through policy-neutral interfaces and provides all system abstractions as optional libraries. However, this flexibility must not imply a lack of security in the system. Denial of service attacks are a serious threat to operating systems since they can compromise the fairness of resource multiplexing, thus favouring some users at the detriment of others, or even preventing the system from functioning at all. Therefore our goal in THINK is to provide the system programmer the elementary tools he needs to build a system in which quality of service can be ensured. However, these tools must remain completely independent of the multiplexing policy chosen by the system programmer. More precisely, the tools must not restrict the programmer's freedom to implement the quality of service policy he deems best for his system. To achieve that, we base our analysis on four hardware resources which should be present in most devices: a processor, some kind of volatile memory (e.g. random access memory), some kind of persistent memory (e.g. a hard drive), and a network. We propose some elementary tools for each resource before detailing the software memory-isolation technique implemented in THINK and presenting a concrete example of the use of these tools.

3.1 Analysis of the four key hardware resources

The processor: Fair sharing of the processor time requires a way to preempt processes, usually by way of interrupts. Since most processors provide some kind of interrupt management, all there is to do is to reify this manager with fitting interfaces. It is also necessary to protect the methods used to register interrupt handlers, to prevent a misbehaving application from illegally installing its own interrupt handler. This can easily be achieved in the THINK framework by securing the bindings and the binding factories. For instance, bindings can be made unforgeable by using cryptography and cyclic redundancy checking, and binding factories can use a standard capability system to check which process can create a binding with a given resource.

Volatile memory: To implement the notion of security domains, we need a way to isolate components or groups of components from each others. Since hardware isolation is usually deemed inflexible and very costly for context switches, we have implemented in THINK a software memory isolation tool, based on segment matching and code

splicing techniques. This tool is implemented as an optional library so as to enable the system programmer to use the hardware isolation provided by the memory management unit of the processor if he prefers so. To permit cross-domains calls, the binding factories can be used to ensure that a component will call (i.e. create a binding with) only authorized methods from other components.

Persistent memory: A fair sharing of the storage space can be easily implemented using classical disk quotas. However, this does not protect against a malicious program which would make repetitive reads and writes of big files to slow down the disk accesses of other applications. Lots of work has been conducted in the field of disk scheduling, basing reordering of disk accesses on criteria like the order of arrival of the request, the localisation of the requested sector on the disk, or the real-time constraints of the request. To protect the system against disk access flooding attacks, these algorithms can be modified to take into account security constraints. For example, a scheduling algorithm can collect statistics on the processes accessing the disk and reorganize the requests so as to treat those coming from identified monopolizing processes after those from normal processes. In THINK, this statistic gathering disk scheduler can easily be programmed as a library component which the programmer can link with his own component in which the disk scheduling policy is defined. Thus, we ensure a complete independence of the policy (implemented by the programmer in his component) and the tool (provided by THINK as an optional library).

The network: Fair sharing of the network bandwidth between the processes is a basic quality of service tool which can be easily parameterized to enforce any chosen bandwidth multiplexing policy and can evolve dynamically as process priorities change. However, the major threat remains the SYN flooding attacks, which have become a common danger for Internet services and especially Web servers. No complete solution has yet been found to protect systems against these attacks but some steps can be taken to lessen the risk of flooding the backlog queue. First, the server can filter packets with source IP addresses obviously forged. A sentry can also count the number of connections in SYN_RECEIVED state and empty the backlog queue before it is saturated. This does not prevent legitimate connections from failing since they will be cancelled when the queue is emptied, but it protects against a backlog queue overflow which might result in a system crash. It is also possible for the server to learn from attacks: if more than a fixed number of SYN packets coming from the same IP address have been discarded because the corresponding ACK packet did not come in time, the server can consider that address to be spoofed and drop all subse-

quent SYN packets coming from it. Finally, SYN cookies [10] can be implemented too, as an optional tool considering the TCP protocol incompatibilities that they induce. In THINK, these tools should be implemented in the binding factories, since flooding a server with connection requests is equivalent with flooding a binding factory with requests for bindings with a remote component.

3.2 A software isolation mechanism

We present here the software isolation mechanism we have developed in THINK to implement the notion of security domains. This mechanism was implemented on a PowerPC processor, but it should be easy to port to any RISC machine.

The algorithm we use to enforce process isolation is based on code-splicing and segment-matching. The goal is to generate code for every memory-access to verify that the access is in an allowed area. This code generation takes place when a process is created at runtime and its code loaded in memory, so as to hide the delay induced by the code parsing and code generation within the process creation delay. When the process code is loaded into memory by the system, the algorithm parses it and generates code to check the address points to an allowed area (this area is identified by its lower and upper limits, which are here held in two dedicated registers). The checking code cannot be directly inserted in the initial code since we are working on raw binary code which would make address translation very difficult and costly. Therefore the generated code is stored in a dedicated memory area and the initial code is modified to replace the memory access by a branch to this generated code. This algorithm can be optimized for instructions with the address encoded in them, like some branches on the PowerPC for example. In that case, the check is made when the initial code is parsed and a process containing an illegal access will not be executed at all.

All the benchmarks we have conducted were executed on a PowerPC G4 866 MHz with 384 MB of SDRAM PC100 memory. All test programs were compiled with `gcc` using the `-O` option for optimizing the memory accesses by putting local variables in registers.

Memory consumption: On the PowerPC, the average size of the generated code for a memory access is 5 instructions. Obviously, not all instructions in a programme make memory access, so we monitored the algorithm for a simple test program to find the average increase. The chosen test algorithm is a bubble sort program, which can be coded in 29 assembly instructions, including 9 memory accesses. Amongst those 9 accesses, 4 are branches those destinations can be statically checked. This results in a generated code size of 25 instructions, which is almost has much as the

original code, therefore doubling the program size in memory. This result can appear to be rather prohibitive, but one must consider that this increase of memory space required only applies to the code of the application, not to its data. Since most applications include much more data than code, doubling the code size is not so costly as it seems.

Code generation delay: We monitored the delay induced by the generation of the spliced code when the process is created in memory. We copied/pasted 1000 times the code of the bubble sort algorithm, therefore obtaining a code of 29000 instructions and found that the algorithm takes 3.90 ms to complete. Since each PowerPC instruction is 4 byte long, we obtain a processing time of 28 MB/s (i.e. the algorithm would take 1 second to modify the original code and generate the spliced code for a 28 MB long original code). Considering that the most consuming part of the process creation task is when the code loader in charge of creating the process accesses the hard drive to read the ELF file, and that the average read time on a standard hard drive is 15 MB/s, we believe that the processing time of the segment matching algorithm is acceptable and will not affect the system performances.

Runtime penalty: We first performed benchmarks for basic operations. First, we monitored the runtime of a single memory access (i.e loading a 32-bit integer from memory in a register) and found that the runtime is multiplied by 3.5 with segment matching. This prohibitive cost is easily understandable, since we add to the runtime of the memory access the runtime of an addition, two comparisons, and two branches. For a local branch, there is no increase since the destination address can be statically checked when the process is created and therefore no code is generated. Finally, for an absolute branch (such are used in inter-process calls), the segment matching induce an increase of +16.67%, which is very competitive compared to the cost of IPC through hardware isolation. We compared this inter-process call with an optimised LRPC [11] and found that the LRPC is more than 25 times slower than our mechanism.

We then monitored the runtime for two significant algorithms. We first sorted 100000 integers with the bubble sort algorithm and found 56970 ms without segment matching and 116280 ms with it, resulting in multiplying the runtime by 2. We then implemented Heron of Alexandria's square root computing algorithm (a classical iterative algorithm), and monitored its performances for 10^6 computations of the square root of 10^6 . The runtime was 6655 ms without segment matching, and 6758 ms with it, thus an increase of 1.55%. We chose these two standard algorithms because the bubble sort is representative of algorithms making lots of memory accesses whereas the square root computing represents algorithms making very few memory accesses. Thus,

we can conclude that the runtime penalty for the software memory isolation algorithm will be below +100%, depending on the frequency of memory access of the application. As an example of a real application, we ported the `gzip` data compression algorithm and found an increase of the runtime of approximately +100%, which is logical since the LZW algorithm makes heavy use of memory access to manage its string table.

Analysis: The prohibitive cost of segment matching memory accesses clearly reduces the interest of our mechanism for enforcing data confidentiality. On the other hand, this mechanism is very interesting for IPC, especially compared to hardware isolation. Thus, combined with a secure framework based on secure binding factories, this mechanism can be used to implement the notion of security domains proposed in the component model on which *Think* framework is based. By isolating component with software memory isolation, we prevent the programmer from directly calling a remote method with a forged pointer for example, thus forcing him to pass through the binding factories where security checks can be made.

3.3 Example

We present here the example of a fair scheduler implemented using the elementary security tools presented above. Considering an application composed of several processes, the programmer wants to ensure that each process will be allocated the same amount of time as the others. The first tool needed is the software memory isolation mechanism, which is used to define a different protection domain for each process, and another one for the scheduler. To be able to interrupt a process execution, the scheduler needs to use the system clock. So it registers a new interrupt handler for that hardware resource using the `TrapRegister` method provided by the interface reifying the interrupts. By doing that, a binding is created between the scheduler and the clock object by the binding factory managing the interrupt handlers. But before creating the binding, the binding factory authenticates the object calling the `bind` method to ensure that it is an allowed object (i.e. the scheduler object and not an application process). Thus, using the tools proposed by the *THINK* architecture, the programmer can ensure that his policy of fair scheduling between the processes will be enforced.

4 Conclusion

As we have shown in this paper, security can be enforced in an operating system without sacrificing flexibility. By providing elementary security tools, the *THINK* architecture

does not restrict the system programmer's liberty to implement whichever security policy suits him best. Combined with a secure framework based on protected binding factories, these tools provide the kernel programmer with all he needs to build a secure customised system.

Bibliography

1. Jean-Philippe Fassinio, Jean-Bernard Stefani, Julia Lawall and Gilles Muller. *THINK: A Software Framework for Component-based Operating System Kernels*. In *Proceedings of the USENIX Annual Technical Conference*, 2002.
2. Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, Olin Shivers. *The Flux OSKit: A Substrate for Kernel and Language Research*. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.
3. Godmar Back, Wilson C. Hsieh, Jay Lepreau. *Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java*. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, 2000.
4. Przemyslaw Pardyak, Brian N. Bershad. *Dynamic Bindings for an Extensible System*. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, 1996.
5. Duane Olawsky, Todd Fine, Edward Schneider, Ray Spencer. *Developing and Using a "Policy Neutral" Access Control Policy*. In *Proceedings of the New Security Paradigms Workshop*, 1996.
6. Olivier Spatscheck, Larry L. Peterson. *Defending Against Denial of Service Attacks in Scout*. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.
7. Gaurav Banga, Peter Druschel, Jeffrey C. Mogul. *Resource Containers: A New Facility for Resource Management in Server Systems*. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.
8. Dawson R. Engler, M. Frans Kasshoek, James O'Toole Jr. *Exokernel: An Operating System Architecture for Application-Level Resource Management*. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.
9. Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham. *Efficient Software-Based Fault Isolation*. In *Proceedings of the ACM SIGOPS'1993*.
10. Jonathan Lemon. *Resisting SYN flood DoS attacks with a SYN cache*. In *Proceedings of the BSDCon 2002 Conference*.
11. Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy. *Lightweight Remote Procedure Call*. In *ACM Transactions on Computer Systems*, Vol. 8, No. 1, February 1990, pages 37-55.

Timing Fault Detection for Safety-Critical Real-Time Embedded Systems

Sébastien Faucou, Anne-Marie Dplanche and Yvon Trinquet
 Institut de Recherche en Communications et Cybernétique de Nantes
 1, rue de la Noë, BP92101 44321 Nantes Cedex 3, France
 {faucou|deplanche|trinquet}@irccyn.ec-nantes.fr

Abstract

On the one hand, a major aspect of dependability for real-time embedded systems is the respect of timing requirements. On the other hand, the complexity of modern real-time embedded system implies the need for new design process focusing on high-level features, such as architecture-based design. In this paper, we show how to integrate a timing fault detection technique in such a design process. Our approach is based upon the CLARA ADL (Architecture Description Language). This language allows to describe applications which can be easily implemented thanks to a distributed middleware designed on top of the OSEK/VDX real-time kernel.

1 Introduction

The reliability and the low-cost of today electronic components have led many industries to increase the part of embedded systems in their field. A representative example is the automotive industry which integrates more and more in-vehicle embedded “Electronic Control Unit” to increase safety and comfort by providing new features. For these embedded real-time control systems, one of the major dependability requirements is safety (the system must not damage its environment). In this paper, we focus on a special kind of safety-related faults: timing faults. For a real-time system, its service is correct not only if the results it delivers have good values but if the dates where they are produced are also good. Actually, in such a context, the violation of a timing constraint can be safety critical to the environment.

Because these systems become more and more complex, they now exhibit more “classical” requirements: flexibility, reuse, interoperability, etc. One way to handle these requirements is to adopt an architecture-based design process, which makes it possible to reason about the architecture level of design [10]. Our goal is to investigate the relations between the architectural design of a real-time appli-

cation and the verification of its timing requirements, and we show how a timing analysis can be conducted at this level. Such an approach, while not requiring a detailed algorithmic knowledge of the application enables to detect and thus to correct design mistakes early in the life cycle. This timing fault detection step is based on a behavioral analysis. Obviously, it does not preclude other means for dependability and particularity fault-tolerance mechanisms.

To perform a timing analysis, the whole system has to be taken into account: application software, execution platform and environment. Our work being based on the CLARA ADL, we explore in a first time the implementation of CLARA descriptions. To facilitate this process, we have developed a dedicated distributed middleware on top of the OSEK/VDX real-time platform. In a second time, we build a fine grain model of this implementation, which takes into account details of the low level software. The simulation of this model allows to observe the timing behavior of the candidate architecture and to validate it w.r.t. the specified timing constraints, that is to detect timing fault occurrences and locate their sources so as to design a new candidate.

2 Context

2.1 In-vehicle embedded systems

Today, vehicles include more and more electronic systems and features. Some of these new functionalities run across different sub-systems which have to communicate. Moreover, this kind of systems is developed for a wide range of products. As a consequence the software and hardware requirements have evolved and include now flexibility, portability, reuse, hardware/software independence, etc.

For the software aspects, one can cite as an illustration the OSEK/VDX architecture [9]. It is a joint project of European car industries the aim of which is to propose a standard platform for in-vehicle embedded applications. It is made up of different specifications: OSEK OS (scalable kernel for real-time embedded systems), OSEK

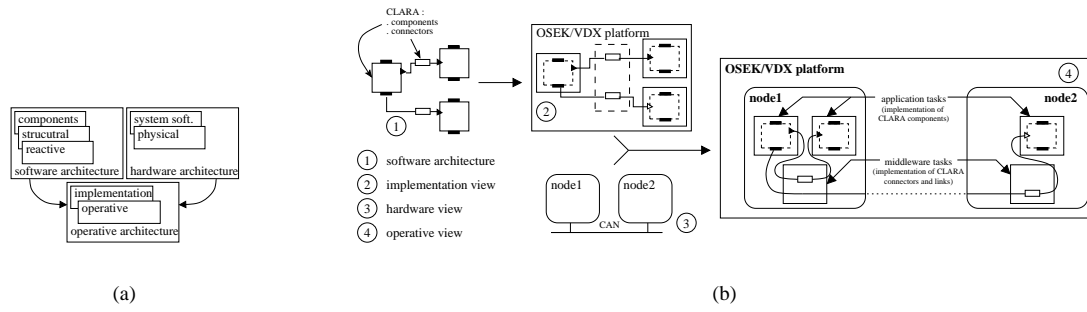


Figure 1. Design process of an operative architecture with CLARA.

COM (application-level communication protocol), OSEK NM (network management), etc.

For the hardware aspects, one of the major requirements is to provide a flexible way to interconnect subsystems. To address this requirement, network protocols have been developed in order to build distributed systems around multiplexed bus that offer an acceptable QoS. These protocols include Controller Area Network (CAN, based on CSMA/CR), Time Triggered Protocol (TTP, based on TDMA), Local Interconnect Network (LIN, master/slaves), etc. To experiment our approach, we will consider systems composed of OSEK/VDX nodes interconnected through a CAN bus.

2.2 Architecture description of real-time applications

Software architecture is a field of interest for scientist since the mid 90's [10]. It is the domain of software engineering dedicated to high-level design. Part of the work on software architecture has been carried out on ADL, which aims at describing a software architecture as the interconnection of *components* (processing entities) and *connectors* (communication entities). Being formally defined, an ADL provides with the possibility to perform architecture-level analysis: liveness [2], reliability [12], etc.

The CLARA ADL, which is dedicated to real-time applications, has been defined in our team [6]. CLARA components are active objects (they own an execution flow) and the set of predefined connectors covers synchronous and asynchronous event signalling and message passing mechanisms. The semantics of CLARA is defined with (time) Petri nets and dedicated static verification techniques allow to perform formal analysis of architecture descriptions. Exhaustive timing analysis techniques are also used, but they do not allow to take into account most of the implementation aspects. As the timing behavior of a system relies heavily on its implementation, this is a severe restriction. We

propose in this paper a simulation-based technique for the timing analysis of CLARA architecture description which takes into account the effective implementation of the system, including hardware and system software details.

CLARA offers different views of software, hardware and operative architectures (resp. SA, HA and OA), each one being relevant for a specific aspect: component (component specification), structural (component interconnection) and reactive views (interface of the environment and components activation mechanisms) for the SA; physical view (description of the physical architecture) and system software description for the HA; implementation and operative views for the OA. The implementation view is the mapping of the SA onto the execution platform native objects and services, whereas the operative view is the mapping of the implementation view onto the physical view. A summary of this organisation is given fig. 1.

To allow a straightforward design of the implementation view, we have defined a set of mapping rules from CLARA high-level artefacts onto OSEK/VDX native objects (e.g. components are mapped onto OSEK OS tasks). Together with these rules, we have designed a middleware which offers high-level services to the application layer for the mapping of connectors and links. To preserve the benefits of the software architecture approach w.r.t. software evolution and reuse, only four services can be used: *Send* and *Receive* for data transmission and *Signal* and *Wait* for event signalling. The knowledge of the architecture allows the middleware to use the mechanisms and data structures corresponding to the specified connector instance. As embedded real-time systems are static (i.e. all objects are known before runtime), the middleware uses static routing tables to perform its services.

For each node in the system, a task is in charge of handling intra-node interactions, delivering incoming messages and sending outgoing messages. It communicates with application tasks using OSEK OS event notification mechanisms and OSEK COM messages (middleware services are

macros based on OS services and behave as transactions). The connectors of the CLARA description are “inlined” in the middleware. This organization is illustrated in fig. 1(b). A prototype has been developed in C.

3 Architecture-level timing analysis

3.1 Related works

Architecture level timing analysis does not come as a replacement for lower-level timing analysis that can be performed once the detailed design step is achieved. It aims at validating early high level design: application partitioning and structure, activation mechanisms, allocation and scheduling of tasks and frames, etc. To perform such an analysis, a narrow approximation of the internal behavior of the components must be known (including estimation of computing times, which can be obtained from a WCET analysis for pre-existing components and by a priori evaluation for other ones).

There is not a lot of architecture-based tool-set dedicated to the design of real-time applications. One can cite the BASEMENT framework [7] for in-vehicle embedded systems and the MetaH ADL [3] for avionics embedded systems. For the timing analysis, BASEMENT comes with an off-line scheduler and a discrete event system level simulator tool. The simulator uses an emulator of the BASEMENT platform which executes the effective code of the components on a modified BASEMENT kernel. Hence, it can only be used after the completion of the detailed design step and is not adapted to early validation of the high level design. To handle timing requirements at high level, the MetaH tool-set includes a schedulability analyzer, which works on analytical models of the tasks and messages sets. These models are refined along the design process but are constrained by the analytical method used. In order to handle more complex task model and to formally verify critical application components (e.g. scheduler, bus driver, etc.) the MetaH team has also been working on linear hybrid automata models. This technology is unfortunately not mature enough to be applied to a whole architecture.

Our goal is to validate the architecture of a system w.r.t. its timing requirements (e.g. basic and end-to-end deadlines, cadence, etc.). It is common that a time-constrained functionality is performed through the cooperation of several tasks, eventually distributed over different nodes. Thus, the middleware, OS services and eventually the networks are involved in its achievement. As the use of OS or middleware services is of high influence with the application (time overhead, rescheduling, update of the status of system objects, etc.), it must be taken into account when analyzing the (timing) behavior of the system. In order to fulfill this goal we adopt a method based on the simulation of a model

of the system built from detailed models of COTS components (e.g. the execution platform) and high level model of application components which are not fully defined. This approach is similar to [5] but can handle a broader range of task behaviors, in order to cover the possibilities of CLARA.

3.2 System modeling

For the modeling and simulation step, we have selected the *ObjectGEODE* framework from Telelogic¹, which is based upon the SDL formal description technique. SDL uses jointly Asynchronous Communicating Finite State Machines and Abstract Data Types paradigms for system modeling, so as to handle concurrency as well as data modeling. It is known to being unadapted to real-time system specification [4] but we use it as a system description language and we are not penalized by its limitations. Moreover, the *ObjectGEODE* simulator tool [11] offers some extensions, especially a semantics which makes it possible to control time progression.

In order to automate the translation of the OA to an analyzable model, we have developed a set of predefined SDL process types: CPU, CAN stack and network, OSEK OS, OSEK COM, interrupt service routine, middleware task, etc. These models include both functional and timing characteristics. To build the system model, they are instantiated and connected following predefined rules. The remaining work resides in the injection of components behavior and low level software configuration in this skeleton. Finally, the system model must be closed by a model of the environment so as to facilitate the simulation.

3.3 Analysis technique and results

For the verification step, *ObjectGEODE* comes with a model-checker. Unfortunately, it does not allow to verify quantitative timing properties (although ongoing works address this [8]). Moreover, model-checking techniques are confronted with the state-space explosion problem when the number of variables in the model is huge. As a consequence, we use presently the tool in simulation mode.

CLARA allows to express the timing constraints of the system on the architecture views. These constraints are translated into observer automata [1] which are merged to the SDL model. When an observed event occurs during the simulation run, the concerned observers change their states and it is thus possible to detect wrong sequences.

As an output, the simulator generates the trace of signals exchanged during the run. From this trace, Message Sequence Charts (MSC) can be generated at different levels of details by selecting the SDL entities of interest. At the highest level, it is possible to isolate only the signals exchanged

¹Telelogic web site: <http://www.telelogic.com>.

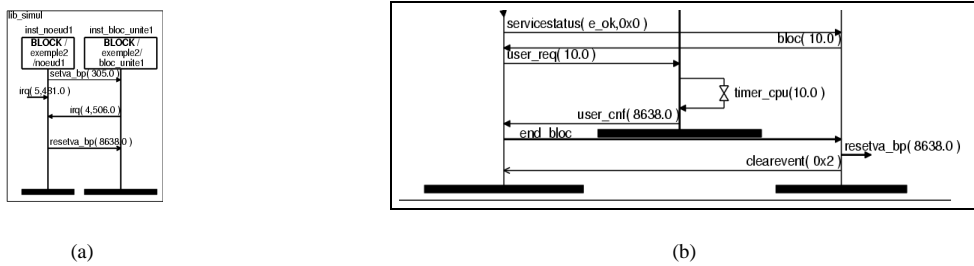


Figure 2. Two MSC generated from the same run at different detail levels.

between a node (seen as a black box) and the environment for a particular functionality. At the lowest level, it is possible to include OS-level signals, especially scheduling signals resulting from middleware and OS service execution: preemption, release, etc.

Figure 2 shows two MSC obtained after a simulation of a factory control system case study. The first (2(a)) is the highest level one: the two entities are node 1 of the control system and the environment. Signals are filtered so as to see only the ones implied in the constrained functionality under study (here the control of valve `va_bp`). This high level MSC underlines a violation of a timing requirement (deadline violation between `irq 4` and `resetva_bp`), detected during the run by the observer associated to this constraint. The second (2(b)) is (a small part of) the lowest level one, the study of which allows to find the source of the timing fault (a priority allocation mistake, allowing a time consuming sampling task to be active for a too long time). To correct this mistake, there are many options the consequences of which have to be analysed by running new simulations: changing the priority assignement, changing the allocation of tasks to node, changing activation mechanisms so as to suppress the conflict, using a simple interrupt service routine to handle the valve, etc.

4 Conclusion

We have exposed an architecture-level timing analysis technique which is usefull to perform early validation of high level design w.r.t. timing constraints. It is based on simulation at system level. For the results to be accurate, the model must be close to the effective system and we adress this issue (i) by offering a middleware and a set of rule for the implementation of architectural artefacts and (ii) by taking into account in the models the knowledge of the runtime platform (hardware architecture, OS, network, middleware, etc.). Aiming at being applied to high level design, it is flexible enough to cover a wide range of real-time application style.

In order to provide with a comprehensive architecture design process, a complementary work should investigate the mapping of event traces collected during the simulation onto the architecture elements. This work should also be directed towards the identification of “good” and “bad” design practices (w.r.t. timing requirements) in order to assist the architect in his corrective work.

References

- [1] B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL : Observing SDL behaviors with GEODE. In *Proc. of SDL Forum 95*. Elsevier, 1995.
- [2] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.
- [3] P. Binns and S. Vestal. Formalizing software architectures for embedded systems. In *Proc. of EMSOFT 2001*. Springer, 2001.
- [4] M. Bozga, S. Graf, L. Mounier, I. Ober, and D. Vincent. SDL for real-time: what is missing. In *Proc. of SAM'2000*, 2000.
- [5] P. Castelpietra, Y. Song, F. Simonot, and O. Cayrol. Performance evaluation of multiple networked in-vehicle embedded architecture. In *Proc. of WFCS'2000*. IEEE IES, 2000.
- [6] E. Durand and A. Déplanche. CLARA - An Architecture Description Language for Real-Time Applications. Technical report, IRCCyN, Nantes, March 1999. (in french).
- [7] H. Hansson, H. Lawson, M. Strömberg, and S. Larsson. BASEMENT: a Distributed Real-Time Architecture for Vehicle Applications. *Real-Time Systems*, 11(3), 1996.
- [8] I. Ober and A. Kerbrat. Verification of Quantitative Temporal Properties of SDL Specifications. In *Proc. of SDL Forum 2001*. Springer, 2001.
- [9] OSEK Group. OSEK/VDX OS 2.2, COM 2.2.2, OIL 2.3, NM 2.5.1. <http://www.osek-vdx.org>, 2001.
- [10] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [11] Verilog. *ObjectGEODE SDL Simulator - Version 4.0*, 1999.
- [12] A. Zarras and V. Issarny. Assessing Software Reliability at the Architectural Level. In *Proc. of ISAW 2000*, 2000.

Towards Trusted Systems from the Ground Up

Vivek Haldar

Information and Computer Science,
University of California, Irvine
CA 92697-3425, USA
vhaldar@uci.edu

Michael Franz

Information and Computer Science,
University of California, Irvine
CA 92697-3425, USA
franz@uci.edu

ABSTRACT

Operating systems, the most fundamental software layer in virtually every computer system, are notoriously insecure and unreliable. A possible reason for this situation is that progress on language-based safety and security mechanisms has largely been ignored in the context of operating systems. There is a lack of *mechanical checking of safety properties* (both at compile- and run-time) as well as a framework and a mechanism for expressing, safely transporting and enforcing such properties. Our solution is to leverage language-based mechanisms by reversing the traditional relationship of operating systems and programming languages – implement operating system functionality on top of a provably safe and secure language and its runtime environment instead of the other way round. We propose to leverage these mechanisms, many of which have been developed in the context of mobile code infrastructures, to build secure systems from the ground up. Such a system would be more secure, flexible and scalable compared to existing systems.

1. INTRODUCTION

The traditional view of computer systems – that is still prevalent today – is that of an operating system abstracting away hardware details and providing basic services such as process management, memory management etc. All applications are built on top of the operating system, utilizing the services it provides. Perhaps the most critical of these applications is the compiler, which translates some high level language to native executable code.

The failing of this model is that even modern operating systems are written in unsafe languages such as C. Buffer overruns and stack smashing are still by far the most common causes of security breaches in modern systems[18]. Trust in an operating system is established with time, as more and more people are able to use it without major problems and more bugs are discovered and weeded out. However, there is no concept of mechanically being able to check the operating system for compliance with a security policy. Thus it is not possible to make any security claims about modern operating systems other than “it’s been out there for two decades and it seems to work”. This problem is starkly highlighted when automated tools such as Engler’s meta-compilation[3] can find hundreds of bugs in operating systems that have been in wide use for a long time and were believed to be relatively bug-free.

At the same time, language-based techniques can now express and check a number of high-level safety properties. Examples of such properties are: “acquired locks must be released[3]”, information flow[23] and “some operations must be done in a certain order[21]”. Also, research in mobile code has developed techniques to *safely transport* these properties as part of some intermediate representation.

In spite of many promising results in language level approaches to security, traditional systems only offer those secu-

rity advantages from the “OS upwards”, because the OS is still in a fundamentally unsafe, weakly typed language (typically C).

The rest of the paper is structured as follows: after giving a very broad introduction to language-based security in section 2, we present our proposed system in section 3. Section 4 presents related work, and section 5 concludes.

2. LANGUAGE-BASED SECURITY AND MOBILE CODE

Language-based techniques for expressing and checking security properties have been gaining importance as a viable and effective way to write secure programs. The case for language-based security has been made elsewhere[15][19] – here we simply give a broad overview of the major techniques and summarize its major advantages.

A *safe* programming language provides certain guarantees about the execution of programs written in that language[16][17]. These include at least the following: *memory safety* – the program reads and writes memory correctly, respecting types, bounds and access restrictions; *control flow safety* – the program does not jump to arbitrary locations (e.g. a location that does not contain code); and *type safety* – operations and function calls receive arguments of the correct type.

In modern programming languages, safety is guaranteed by a combination of static and dynamic checks. *Type systems*[16] are an extremely effective method to rule out a large number of run-time errors using a method that is well understood, has a solid theoretical foundation and where checking is efficient in practice.

Lack of safety in programming languages is a major source of security violations in computer systems. A survey of various well-known bug tracking resources shows that almost half of all exploited vulnerabilities in software can be blamed on buffer overruns[18].

The *language-based*[15][19] approach to security has been variously defined as “a set of techniques based on programming language theory and implementation...brought to bear on the security question”, and “leveraging program analysis and program rewriting to enforce security policies”. These techniques can be thought of as falling into two major categories – program analysis and program rewriting.

Program analysis covers a variety of techniques that statically try to check a program’s conformance to a security policy. Examples are types-based approaches to security such as typed assembly language (TAL)[20].

Program rewriting is a complementary set of techniques that enforce security policies by rewriting programs to conform to them. Inlining security monitors[21] is an example of this. Another example is proof-carrying code (PCC)[14] in which the compiler, along with object code for a program, emits a *certificate* of proof for conformance with a security policy. Though such proofs might be hard to generate, they are very easy to verify.

Each approach has its advantages and disadvantages, and a comprehensive, flexible, expressive and powerful security architecture would need to combine all three elements in a thoughtful manner.

The primary advantage of language-based security is that it is flexible and can easily express very fine-grained policies. For example, the memory protection policy of a type-safe program is very fine-grained compared to the coarse page-level protection that modern operating systems typically provide. Also, since the programmer deals with language-level semantics while writing a program, it is much more useful and intuitive to raise security concerns to the language-level rather than leave them in other parts of the system.

To transport secure code between machines, we need to encode programs in a portable, safe, and tamper-proof manner; these goals are identical to those of mobile code infrastructures. Mobile code has been an extremely active area of research spurred by the ubiquity of the Internet and the consequent need for running third-party untrusted code. One outcome of this research has been a range of techniques for ensuring the *security* of mobile code, especially by using specialized *mobile code formats*. These mobile code formats are amenable to security checks. Thereafter, they could be interpreted or compiled to native code for execution. Examples are the popular bytecode-based Java format[24], as well as other higher level representations such as abstract syntax trees[25]

Note that even though secure mobile code formats were developed in the context of running untrusted code over the Internet, their security advantages are just as attractive for running *any* code. In fact, we propose that in order to have a truly trusted system, we should do away with native code altogether, and *all* code in the system should be in some secure format. Native code would only be generated when needed.

3. A WHOLE SYSTEM BUILT ON SECURE CODE

Our goal is to build a practical system about which we can make security guarantees from the *ground up*, and not just from the operating system up. We assume that all hardware (or at least the CPU) is part of the trusted computing base, and start from there. We propose making use of language-based security to build a trusted system from the level of hardware and above.

We propose to do this by essentially inverting the roles of compiler and operating system in the traditional view of computer systems. Instead of the compiler and other applications being conceptually built on top of the operating system, we now have the compiler as the *first layer* above hardware. The function of this compiler is to translate some secure, type safe intermediate representation (IR) to native code. The *whole system*, including the operating system, is now built on top of this compiler. It remains to be investigated what intermediate representations are suitable for this, and whether it would be best to have one intermediate representation or support a number of them. It also remains to be investigated what the right mix of static and dynamic checks is.

The proposed system would utilize a compiler to compile high-level programs down into a secure type-safe intermediate representation. A second compiler, which essentially sits in the place that traditionally would be occupied by the operating system, would translate this intermediate representation code to native machine code at the time of execution. Note that that for most secure intermediate representations only the sec-

ond compiler (from IR to native code) needs to be part of the trusted computing base.

The advantage of this approach is that it enables us to leverage all the existing research on language based security and apply it to the design of a secure system from the hardware up. When the whole system is built on top of a secure type-safe IR, it is possible to reduce most security concerns to the language level, where the compiler can automatically check them. A pleasant by-product of this is *portability* since all code is in some machine-independent intermediate representation.

One challenge here is to build this system with the absolute minimum of unchecked code that has to be trusted, while retaining acceptable performance.

3.1 A Typed View of Hardware

We also propose to carry types down to the level of hardware by exposing the hardware to the upper layers of the system in a *strongly typed* manner. We call this a typed hardware abstraction layer or *typed-HAL*. This is in keeping with our philosophy of using language-level security mechanisms to the fullest. By exposing hardware only as a typed interface, and *disallowing arbitrary operations on it*, it becomes all the more difficult to subvert the system. Moreover, it does away with a large number of potential bugs in systems that deal with hardware in a raw, untyped manner.

For example, consider the common activity of handing down a network packet for transmission to the network card in a PC. The network card essentially views this packet as a *raw sequence of bytes*. However, note that this packet is *not* just a raw sequence of bytes – it is required to have a very particular structure, as dictated by the particular network protocol being used, say TCP/IP. In modern operating systems, *no check is done to make sure the packet has this particular structure*. The fact that the packet is well formed is taken for granted because it was formed by another part of the operating system, which is trusted¹. Packets that are intentionally constructed to be malicious may exploit loopholes. One conspicuous example of this was the Ping Internet exploit (also called “The Ping of Death”[13]) that could crash a remote machine simply by sending a specially constructed packet to it. There are no automatic mechanical checks. Essentially, we trust the programmer who implemented the network protocol stack of the operating system.

Such problems would vanish in a system that took a typed view of hardware and the data that is sent in and out of it. In the example given above, had the operating system and network protocol stack been written in a strongly typed language, it would be possible to define the structure of the packet as a type. Combined with a typed-HAL, that would only expose the network card as a *typed interface*, we would be able to easily do away with bugs in the structure of network packets by virtue of strong type checking *that is automatically done by the compiler*.

For another example, consider the problem of buffer overruns, which is by far the most common cause of security breaches. Strongly typed language runtimes would prevent such exploits. Strong typing would not prevent a programmer from indexing an array outside bounds, but it would catch the illegal access at runtime.

If we ask the question – *what is the basis for insecurity in a system?* – The answer is deceptively simple. Fundamentally,

¹ The problem is even more severe when third party modules are inserted into the kernel to extend it.

insecurity arises from *doing operations that are not supposed to be done*. Then ask the question – *what are types used for?* – And the answer is: *making sure that only allowed operations are performed on data items of the correct type*. From these two observations it is easy to see the motivation for proposing a typed hardware abstraction layer. By exposing the hardware *only* as a typed interface, we seek to disallow arbitrary operations and limit the operations possible by using hardware to only the legal ones. This does away with a major source of security breaches in one clean swoop. In essence, the programmer sees only a typed view of the machine. This model of the machine is already familiar to programmers using typed high-level languages such as Java.

4. RELATED WORK

Our approach is similar in spirit to a number of operating system projects that built entire operating systems using type safe languages.

SPIN[1] is an operating system written in Modula-3. It uses Modula's type safety and encapsulation properties to enforce safety, modularity and protection between applications. Extensions to the OS are also written in Modula. This makes it possible to add extensions to the operating system in a safe manner, because they are type checked at compile-time, which implies their safety. Type safety and encapsulation are also used to enforce separation between logical domains by using separate namespaces, a language-level feature. However, the main objective of the designers of SPIN was to create an operating system that could be safely extended to meet the specialized requirements of applications. As such, the use of Modula's safety features is relied upon only for extensions within the kernel, and not for the system as a whole.

The Oberon system[2][10] is an entire system written in the language of the same name. Its main emphasis was on extensibility of the system. Safety was guaranteed by the fact that everything in the system was written in Oberon, a strongly typed language. The Oberon system turned out to be surprisingly portable and was implemented on a large variety of hardware platforms.

In current mainstream operating systems such extensibility is usually through some sort of module or device driver mechanism that can add new functionality to the kernel or support a new hardware device. But these modules are not checked for safety and are essentially trusted by the operating system. Since these modules run inside the kernel in privileged mode they can access any data in the system. A malicious module has free reign in the system. A bug in the module affects the entire system. If a module crashes it takes the entire system with it.

In the Oberon and SPIN systems, safety is attained by virtue of using a type-safe language. This is what makes it safe to insert extensions into the operating system. However, this has the disadvantage of adding the compiler for that language to the trusted computing base, because we must trust the compiler to emit code that conforms to the type system of the language and has been checked properly.

A number of systems, such as JavaOS[26] and JX[27], have implemented a Java virtual machine on bare hardware, with some OS-like functionality such as processes.

Our approach is much broader, since we propose using the full gamut of static as well as dynamic checks to express and enforce high-level safety and security properties, and not just those expressible in the Java bytecode model. We would also like to explore the design space for mobile code representations beyond bytecode.

In another approach, safety of extensions is ensured not by type-safety of a language but by a combination of software fault isolation[9] and transaction monitoring. An example of this is the Vino system[6][7], which uses software fault isolation to enforce safety of memory accesses. In order to prevent an extension (which in the Vino system is called a *graft*) from unduly holding resources their execution is treated as a sequence of *transactions* that the kernel keeps track of. Whenever an extension oversteps its bounds, (be it holding the CPU for too long, or allocating too much memory) the kernel terminates its execution and simply unwinds to the state at the time of the last transaction. This technique has a considerable overhead and slowdowns of up to 200% have been reported[7].

Note that the checks that are done by software fault isolation become unnecessary in a strongly typed language. As the system designers of Vino themselves concede, using a typed language would have saved them much implementation effort and would probably have resulted in a more efficient system.

The Exokernel project[4] is based on the assumption that operating system abstractions get in the way of efficient implementation of applications. This is because operating system abstractions are designed to be general and more often than not are not a good match with the specialized requirements of a particular application. This can lead to poor performance. The aim of an exokernel is to provide a very thin, minimal abstraction of hardware, as devoid of policy as possible. The operating system is then simply reduced to a user-level library. The major performance gain comes from eliminating most of the overhead of context switches between user and kernel mode, and improvements of up to four times have been reported for typical applications.

Our proposal would share the advantages of the exokernel approach, since we also propose exporting a very thin, minimal interface of hardware to the rest of the system. But in addition, one of our primary goals is security, and unlike the exokernel project, we propose exporting a typed view of hardware.

The Flux research group at the University of Utah has built a modular, component-based toolkit for building operating system, called OSKIT [5]. Many language researchers have used OSKIT to port implementations of various high level languages to run directly on hardware, as opposed to running them on top of an existing operating system. Their primary goal, however, is to explore how high level language level mechanisms (such as continuations [8]) can be efficiently implemented on hardware, without a policy-laden operating system getting in the way. Also, since OSKIT itself is implemented in C, it suffers from the same problems that we pointed out.

An example of using types for enforcing security properties is *packet types*, proposed independently by Sekar et al [12] and Chandra et al[11]. They use the concept of types to check the structure of network packets. They define a small domain-specific language for defining the structure of network packets. This language is strongly typed. It uses inheritance to capture the idea of protocol stacks with nested packet structures. Packet definitions in this language are used to automatically generate code that can parse and check incoming packets for conformance. An added advantage is the ability to concisely do pattern matching on packets. Sekar et al use this to detect low-level network attacks[12]. Though they use a domain-specific language, most of the notions used can be carried over easily to modern strongly typed languages such as Java. However, some special runtime support would be needed

5. SUMMARY

We note that most security breaches in modern computer systems are a consequence of weak typing. This is because almost all modern operating systems are implemented in a weakly typed language such as C in which most checks are left to the programmer, and are not mechanically enforced. The field of language-based security offers many promising solutions to the problem of specifying security policies, checking conformance against them and enforcing them.

We propose to build a trusted system on tamper-proof trusted hardware, from the ground up, by leveraging a combination of language-based techniques for security. We envision all code in the system, including the operating system, being in some type-safe, secure intermediate representation. At the heart of our system is the idea of bringing language-based security into the operating system kernel itself, eliminating a substantial source of security breaches. We propose to do this by implementing on bare hardware a compiler for a secure type-safe intermediate representation. The rest of the system would use the abstractions provided by this compiler kernel. Also, we propose to export only a strictly typed view of hardware to the rest of the system. We call this a typed hardware abstraction layer. This again eliminates another important source of security breaches, by constraining the ways in which hardware can be used.

We see the following as novel contributions of the proposed research:

Security from the ground-up: starting from hardware that is trusted, building a secure system from the ground up, and not just from the operating system-up, as is the case with current systems.

Typed hardware abstraction layer: exporting only a typed view of hardware to the rest of the system, which would automatically disallow illegal operations on hardware and ensure data integrity.

Building a whole system from mobile code: leveraging language based security right from the level of hardware, and implementing the entire system, including the operating system in a secure type safe intermediate representation, enabling us to make safety guarantees about it.

ACKNOWLEDGMENTS: Thanks are due to Niall Dalton, Peter Froehlich, Peter Housel, Fermin Reig, Christian H. Stork and Alex Strashny for many fruitful discussions and comments.

6. REFERENCES

- [1] B. Bershad, S. Savage, P. Pardyak, E. Gunder, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 267–284, 1995.
- [2] M. M. Brandis, R. Crelier, M. Franz, and J. Templ. The Oberon System Family. *Software—Practice and Experience*, 25(12):1331–1366, Dec. 1995.
- [3] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, 23–25 Oct. 2000.
- [4] D. R. Engler, M. F. Kaashoek, and J. O’Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.
- [5] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux OSKit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997.
- [6] Y. E. James. VINO: The 1994 fall harvest.
- [7] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Operating Systems Design and Implementation*, pages 213–227, 1996.
- [8] O. Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Second ACM SIGPLAN Workshop on Continuations*, Jan. 1997.
- [9] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, 1993.
- [10] N. Wirth and J. Gutknecht. *Project Oberon*. Addison-Wesley, 1992.
- [11] S. Chandra and P. J. McCann. Packet Types. In *Second Workshop on Compiler Support for Systems Software (WCSSS)*, May 1999.
- [12] R. Sekar, Y. Guang, S. Verma and T. Shanbhag. A High-Performance Network Intrusion Detection System. In *ACM Symposium on Computer and Communication Security*, 1999.
- [13] CERT Advisory CA-1996-26. Denial-of-service attack via ping. <http://www.cert.org/advisories/CA-1996-26.html>. October 1996.
- [14] G. C. Necula. Proof-carrying code. In *Conference Record of POPL’97: The 24th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 15–17, 1997.
- [15] D. Kozen. Language-based security. In *Mathematical Foundations of Computer Science*, pages 284–298, 1999.
- [16] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002. To be published.
- [17] L. Cardelli. Type systems. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*. CRC Press, 1996.
- [18] D. Wagner, J. S. Foster, E. A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego, CA, pages 3–17, February 2000.
- [19] F. B. Schneider, J. G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics*, pages 86–101, 2001.
- [20] G. Morrisett, D. Walker, K. Cray, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.
- [21] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop*, pages 87–95, Ontario, Canada, 22–24 September 1999.
- [22] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN ’01 Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices*, 36(5), May 2001.
- [23] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Conference Record of POPL’99: The 26th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages [POP99]*, pages 228–241.
- [24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
- [25] C. H. Stork and V. Haldar. Compressed Abstract Syntax Trees for Mobile Code. In *Workshop on Intermediate Representation Engineering (IRE 2001)*, July 2001, Orlando, Florida.
- [26] T. Saulpaugh and C. Mirho (1999) *Inside the JavaOS Operating System*. Addison Wesley, Reading, Massachusetts.
- [27] M. Golm, J. Kleinöder, and F. Bellosa. Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System. In *HotOS 2001*, May 20–23, 2001, Elmau/Oberbayern, Germany.

Studying and using failure data from large-scale Internet services

David Oppenheimer and David A. Patterson

University of California at Berkeley, EECS Computer Science Division

387 Soda Hall #1776, Berkeley, CA, 94720-1776, USA

{davidopp,patterson}@cs.berkeley.edu

Abstract

Large-scale Internet services are the newest and arguably the most commercially important class of systems requiring 24x7 availability. As a result, very little information has been published about their causes of failure. In an attempt to address this deficiency, we have analyzed detailed failure reports from three large-scale Internet services. Our goals are to (1) identify the major factors contributing to user-visible failures, (2) evaluate the (potential) effectiveness of various techniques for preventing and mitigating service failure, and (3) build a fault model for service-level dependability and recovery benchmarks. Our initial results indicate that operator error and network problems are the leading contributors to user-visible failures, that failures in custom-written front-end software are significant, and that online testing and more thoroughly exposing and handling component failures would reduce failure rates in at least one service.

1. Introduction

The number and popularity of large-scale Internet services such as Google, MSN, and Yahoo! have grown significantly in recent years. Moreover, such services are poised to increase in importance as they become the repository for data in ubiquitous computing systems and the platform upon which new global-scale services and applications are built. These services' large scale and need for 24x7 operation have led their designers to incorporate a number of techniques for achieving high availability. Nonetheless, failures still occur.

While the architects and operators of these services might see such problems as failures on their part, these system failures provide important lessons for the systems community about why large-scale systems fail, and what techniques are or would be effective in preventing component failures from causing user-visible service failures. In an attempt to answer the question "Why do Internet services fail and what can be done about it?" we have studied the architecture of, and 62 post-mortem reports of user-visible failures from, three large-scale Internet services. In this paper we describe three initial directions for using this

data. First, we identify which service components are most failure-prone, so that service operators and researchers know what areas most need improvement. Second, we examine the applicability of a number of failure mitigation techniques to the actual failures we observed. Third, we suggest using the failure data to derive a fault model for service-level dependability and recovery benchmarks.

2. Survey sites and methodology

We studied an online service/Internet portal (*Online*), a global content hosting service (*Content*), and a high-traffic, read-mostly Internet service (*ReadMostly*). Physically, all of these services are housed in geographically distributed colocation facilities and use commodity hardware and networks. Architecturally, they are built from a load-balancing tier, a stateless front-end tier, and a back-end tier that stores persistent data; and they use multiple levels of redundancy and load balancing for performance and availability. Operationally, they use primarily custom-written software to provide and administer the service; they undergo frequent software upgrades and configuration updates; and they operate their own 24x7 Systems Operations Centers staffed by operators who monitor the service and respond to problems.

The services we examine differ slightly in their workloads and node hardware. *Online* and *ReadMostly* each receive about 100 million hits per day, while *Content* receives about 7 million. The ratio of writes to reads in *Online* and *Content* is moderate, while that in *ReadMostly* is (as the name implies) low. Finally, all three services use x86-based PCs running open-source operating systems throughout their service, except for *Online* which uses Network Appliance file servers for back-end storage.

Because we are interested in *why* and *how* large-scale Internet services fail, we studied individual problem reports rather than aggregate availability statistics. The operations staff of all three services use problem-tracking databases to record information about component and service failures. Two of the services (*Online* and *Content*) gave us access to these databases, and one of the services (*ReadMostly*) gave us access to the problem post-mortem reports written after every major user-visible service fail-

ure. For *Online* and *Content*, we defined a user-visible failure as one that theoretically prevents an end-user from accessing the service or a part of the service (even if the user is given a reasonable error message) or that significantly degrades a user-visible aspect of system performance¹.

We studied 16 failures from *Online*, and 23 from each of *ReadMostly* and *Content*. These problems corresponded to the user-visible failures during four months at *Online*, six months at *ReadMostly*, and a month at *Content*. Note that it is not fair to compare the services directly, as the functionality of the custom-written software at *Online* is richer than that of *ReadMostly*, and *Content* is more complicated than either of the other two services (e.g., *Content* counts as service failures not only failures of equipment in its colocation sites, but also failures of client proxy nodes it distributes to its users, including situations in which those client proxies cannot communicate with the colocation facilities). But because we studied all user-visible failures for each service, and used approximately the same definition of failure for choosing (or having chosen for us) the problems to examine from each of the services, we believe our conclusions as to relative failure causes are meaningful.

We attributed the cause of a system failure to the *first component that failed* in the chain of events leading up to the service failure, and the nature of that component's failure to the *type of flaw (fault) that was the root cause of the failure*. In particular, the failing component was categorized as **front-end node**, **back-end node**, or **network**, and the underlying cause of the failure as **hardware**, **software**, **environment**, **operator error**, or **unknown** (indeterminable). Note that the underlying flaw may have remained latent for an arbitrary period of time, only to cause a component to fail when another component subsequently failed or the service was used in a particular way for the first time. *Front-end nodes* are those initially contacted by end-user clients, as well as the client proxy nodes used by *Content*. Using this definition, front-end nodes do not store persistent data (though they may cache data), while back-end nodes do store persistent data. The "business logic" of traditional three-tier systems terminology is therefore part of *front-end*, a reasonable decision because these services integrate their service logic with the code that receives and replies to user client requests.

¹ "Significantly degrades a user-visible aspect of system performance" is admittedly a vaguely-defined metric. A more precise definition of failure would involve correlating component failure reports with degradation in some aspect of observed system performance such as response time. But even where these services measured and archived response times for the time period studied, we are not guaranteed to detect all user-visible failures, due to the periodicity and placement in the network of the probes. Thus our definition of *user-visible* is problems that were *potentially* user-visible, i.e., visible if a user tried to access the service during the failure.

Most problems were relatively easy to map into this two-dimensional component-flaw space, except for wide-area network problems. Such problems affected the links between colocation facilities for all services, and, in the case of *Content*, also between customer sites and colocation facilities. Because the root cause of such problems often lay somewhere in the network of an Internet Service Provider to whose records we did not have access, the best we could do with such problems was to label them as **network** and due to a flaw of **unknown** cause.

3. Analysis of problem causes

Classifying the 62 problems we reviewed has allowed us to make a number of observations about the causes of user-visible service failures. The data from which we make these observations are summarized in Table 1, which breaks down problem causes by the part of the service containing the root cause, and in Table 2, which breaks down problem causes by the component that failed and the underlying cause of the failure.

Table 1 shows that contrary to conventional wisdom, front-end machines can be a significant source of failure. In the services we studied, this was largely due to the complexity of the service software running on those machines and the complexity of configuring and administering them.

Table 2 shows that operator error and networking problems are a significant cause of failure. In *Online* and *Content*, operator error caused more failures than did node hardware or software failures, while in *ReadMostly* networking problems caused more failures than did node hardware or software problems. Networking failures were prominent because networks tended to be a single point of failure--services often use only one network switch to connect their server racks to the colocation site's network, and colocation facilities often used only one Internet Service Provider (indeed, such facilities are often owned and operated by an ISP). This latter fact means that even colocation sites with multiple physical Internet links may be adversely impacted by a single upstream Internet failure. We also observed that while geographic redundancy tends to reduce the incidence of complete service unavailability, many Internet problems nonetheless become user-visible

	front-end	back-end	network
Online	63%	25%	13%
Content	57%	17%	26%
RdMostly	4%	9%	87%

Table 1: Failure cause by part of service.

because of non-fail-stop failure modes of Internet links, and delays in detecting a problem and then updating global load balancing tables. Colocation facilities did appear effective in eliminating “environmental” problems—only one environmental problem in our study led to a user-visible failure, and that problem was a power failure at one of *Content*’s customer sites, not at a colocation site.

4. Techniques for mitigating failure

Given that user-visible failures are inevitable despite these services’ attempts to prevent them, how could the failures have been avoided or their impact reduced? To answer this question, we analyzed each of the 62 problem reports, asking whether any of a number of techniques that have been suggested for improving availability could potentially

- prevent the original **component fault** (*e.g.*, a double-bit memory error, a software bug, an error in a configuration file, or an incorrect operator command),
- prevent a component fault from turning into a **component failure**,
- reduce the severity of degradation in user-perceived system quality of service (QoS) due to a component failure (*i.e.*, reduce the degree to which a **system failure** is observed),
- decrease the **TTD**: time from component failure to detection of the failure,
- decrease the **TTR**: time from component failure detection to component repair (*i.e.*, the time during which system QoS is degraded).

The above categories can be viewed as a state machine or timeline, with component fault leading to component failure, causing a user-visible system failure; the component failure is eventually detected and repaired, returning the system to its failure-free QoS.

The techniques we investigated for their potential effectiveness were

- **redundancy**: replicating data, computational functionality, and/or networking functionality [2]
- **isolation/partitioning**: increasing isolation between software components, to reduce failure propagation
- **restart**: periodic rebooting of hardware and restarting of software
- **fault injection and load testing**: explicitly testing failure-handling code and system response to overload by artificially introducing failure and overload scenarios, either into components before deployment or into the production system
- **testing**: testing the system for correct behavior given normal inputs, either in components before deployment or in the production system
- **config**: using tools to check that low-level configuration files meet sanity constraints
- **exposing**: better exposing software and hardware component failure to other modules and/or to a monitoring system

Table 3 shows the number of problems from *Online*’s problem tracking database for which use, or more use, of each technique could potentially have prevented the problem that directly caused the system to enter the corresponding failure state. A given technique generally addresses only one or a few system failure states; we have listed only those we consider feasible.

Note that if a technique prevents a problem from causing the system to enter some failure state, it also necessarily prevents the problem from causing the system to enter a subsequent failure state. For example, checking a configuration file might prevent a component fault, which therefore prevents the fault from turning into a system-level failure, a degradation in QoS, a need to detect the failure, and a need to repair the failure. However, our methodology only counts this as preventing a component fault, so as to more precisely pinpoint the effect of the technique. Finally, note that techniques that reduce time to detect or time to repair component failure reduce the overall service

	node op	net op	node hw	net hw	node sw	net sw	node unk	net unk	env
Online	44%	0%	13%	6%	31%	0%	0%	6%	0%
Content	35%	4%	0%	0%	26%	0%	9%	22%	4%
Read-Mostly	13%	9%	0%	17%	0%	26%	0%	35%	0%

Table 2: Failure cause by component and fault type. The component is described as node (node) or network (net), and fault type is described as operator error (op), hardware (hw), software (sw), unknown (unk), or environment. Operator and network failure are the leading causes of service failure.

loss experienced (we define the loose notion of “overall service loss” as the amount of QoS lost during the failure, multiplied by the duration of the failure).

From Table 3 we observe that a large number of the problems *Online* experienced might have been prevented or mitigated by more online testing, increased redundancy, and more thoroughly exposing and reacting to software and hardware failures. Automatic sanity checking of configuration files, and online fault and load injection, also appear to offer significant potential benefit.

Additional results from the three services, including an analysis of time-to-repair for the various types of failures, the causes of non-user-visible failures, and lessons from individual problem case studies, can be found in [3].

technique	system failure state avoided/mitigated	# of instances potentially avoided
redundancy	system failure	8
isolation/part.	system failure	2
restart	component fail	1
pre-fault/load	component fault	2
online fault/load	component fail	3
pre-testing	component fault	1
online testing	component fail	11
config	component fault	3
expose/monitor	TTD	8
expose/monitor	TTR	9

Table 3: Potential benefit from using in *Online* various proposed techniques for avoiding or mitigating service failures. Nineteen problems were examined (rather than sixteen as in Section 3) because here we have also included problems whose sources were external to the service (i.e., due to failure in an Internet site that *Online* uses to provide part of its service). Pre-fault/load refers to fault injection and load testing prior to system deployment, while online fault/load refers to such testing conducted in a production environment. Pre-testing and online testing having similar meanings, but for correctness testing. Those techniques that *Online* is already using are indicated in bold; in those cases we evaluate the benefit from using the technique more extensively.

5. Fault models for service-level benchmarks

In addition to indicating where to focus efforts for improving availability, and helping to evaluate the potential effectiveness of specific techniques, the failure data we have collected can be used to create a fault (or, to use our terminology, component failure) model for *service-level benchmarks*. Recent benchmarking efforts have focused on component-level dependability by observing single-node application or OS response to misbehaving disks, system calls, and the like. But because we found a significant contribution to service failure of human error (particularly multi-node configuration problems) and network (including WAN) problems, we suggest a more holistic approach. In *service-level benchmarks*, a small-scale replica (or a physically or virtually isolated partition) of a service is created, and Quality of Service for a representative service workload mix is measured while representative component failures (e.g., those described in this paper) are injected. To simplify this process, one might measure the QoS impact of individual component failures or multiple simultaneous failures, and then weight the degraded QoS response to these recovery events by either the relative frequency with which the different classes of component failure occur in the service being benchmarked, or using the representative proportions we found in our survey. As suggested in [1], the human operator role in both causing and repairing failures, and in conducting normal service administrative tasks, should be included.

6. Conclusion

From a study of 62 user-visible failures in three large-scale Internet services, we observe that front-ends are a more significant problem than is commonly believed, that operator error and network problems are leading contributors to user-visible failures, and that more thoroughly exposing and handling component failures would reduce failure rates in at least one service. Because human error and network problems dominate, we argue for *service-level benchmarks* that replicate a service’s hardware and software architecture, its component dependencies, its workload, its failure modes, and its human operator tasks.

References

- [1] Brown, A., L. C. Chung, D. A. Patterson. Including the Human Factor in Dependability Benchmarks. *2002 DSN Workshop on Dependability Benchmarking*, 2002.
- [2] J. Gray. Why Do Computers Stop and What Can Be Done About It? *Symposium on Reliability in Distributed Software and Database Systems*, 1986.
- [3] D. Oppenheimer. Why do Internet services fail, and what can be done about it? UC Berkeley Technical Report UCB-CSD-02-1185, 2002.