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;

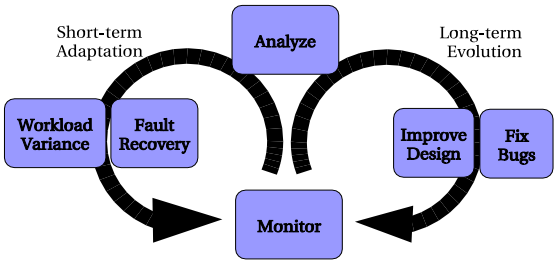

Figure 1. Short-term and long-term feedback loops

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 messeging, 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. Kubiatowicz, 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. Kıcıman, 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. Kubiatowicz, 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 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.