# Reflections on the History of Operating Systems Research in Fault Tolerance

**Ken Birman**
**Dept. of Computer Science, Cornell University**
**http://www.cs.cornell.edu/ken**

## Abstract.

In October of 2015, we celebrated the 50th anniversary of SOSP as part of SOSP 25. Peter Denning formed a "history day" steering committee, and invited me to give a short talk on the topic of fault tolerance (and also asked if I could help organize the remainder of the day). This essay is intended as an accompaniment to the video and slides of my talk.

My topic here is dominated by two fairly specific questions, central to the way we think about the discipline: Must strong properties bring complexity, poor scalability, high latencies and other significant costs? Can a system support consistency without dictating to its users? The debate surrounding these issues has animated the community at least since the mid 1980's.

Any decision to focus dictates a degree of narrowness. For example, Butler Lampson has argued that one cannot have security without reliability, and vice versa [25], and the SOSP 2015 program supports his view. Nonetheless, I won't be discussing security here. I hope that nobody is offended by my omission of this and other important work; just like the other history day speakers, I was required to keep the scope of my talk manageable and focused, and omissions were unavoidable.

## Introduction.

Fault tolerance has been important to the operating systems community from its earliest days, but the term has completely different meanings within distinct subsets of the community. In this essay, I'll touch upon several of those many meanings, but for brevity and clarity will focus on a more restricted question: *Is fault tolerance (specifically, approaches that use consistent data replication) at odds with the most basic principles of the operating systems field and community?*

That one might ask such a question may seem perplexing to those new to our field: if you explore the program for SOSP 2015, you'll quickly see that fault tolerance has become one of the most dominant themes: six papers are concerned with Paxos, and another three with transactional mechanisms, and beyond those nine, others explore consistency and correctness after failures. In 2015, the SOSP community is very much a fault tolerance and consistency community.

However, this was not always so. In 1993 SOSP was torn by a huge debate associated with a paper by Cheriton and Skeen entitled *Understanding the Limitations of Causal and Totally Ordered Communication* [1]. What we've come to refer to as the CATOCS controversy [2][3][4] centered not so much on whether one could build communication tools and platforms that offer strong fault tolerance and consistency guarantees, but whether in doing so one arrived at application-specific mechanisms, not belonging in the operating system. The underlying theme was that consistency mechanisms don't perform or scale adequately to belong to the core systems area. Hence, the "S" in CATOCS.

The 2015 SOSP program represents one form of judgement on that question, but in some sense, isn't a direct response to the question: none of those papers were really focused on core operating systems components or advocating new principles, and none referred back to the CATOCS controversy. Indeed, as of 2015, the CATOCS argument itself had reemerged in a new form.

The modern version is associated with Eric Brewer's CAP conjecture, capturing a line of thought that I first became aware of after Eric gave a keynote talk at PODC in 2000 [5] and then presented his SEDA paper at SOSP in 2001 [6]. Eric suggested that there may be deep tradeoffs between *consistency* (by which he meant database-style ACID guarantees), *availability*, and *partition tolerance* of large-scale systems. Inktomi, a scalable system Eric and his group created for web search, leveraged CAP to gain better performance and scalability, and Eric argued that other web-based systems could do so as well. Later, a more general version of CAP came to be widely adopted by the cloud computing community (today we might call this the PaaS community: Platform as a Service, which is one of the main styles of using the cloud, and refers to applications created on subsystems like Google's AppEngine or Microsoft's Azure platform). CAP shaped several SOSP papers too, notably the Amazon Dynamo paper in 1997 [7], and eBay's recommendation that cloud developers reject ACID and embrace BASE [8]. A version of CAP was soon proved in a paper by Gilbert and Lynch [9], although for a fairly narrow scenario.

CAP has become the emblem of an outspoken community that builds big cloud platforms and believes that strong forms of consistency and fault tolerance are at odds with scalability, fast response and overall system capacity. Whereas CATOCS never won a large following, CAP has succeeded in this sense. Yet how can an embrace of inconsistency not sound reckless and wrong? At the very least, we as a field really should try to understand the underlying rationale. Yet doing so isn't trivial: CATOCS and CAP are both somewhat ambiguous concepts, and even the proponents aren't entirely clear about what these principles really mean, or why the BASE methodology is sound. The CAP community seems to think of CAP as a very broad and universal principle, and yet the Gilbert and Lynch paper points out that their CAP theorem wouldn't hold if the definitions are relaxed in any substantial way, and even offer an example of a practical scenario under which one really can have all three properties at once.

To me this highlights a tension within our community. Many practitioners, working down in the trenches, have become convinced that *cloud-scale systems just can't afford consistency*. Meanwhile, the 2015 SOSP community seems to believe that CAP is just a practical obstacle that can be surmounted with clever systems work. Which perspective is the right one? The answer isn't completely obvious, because the SOSP community isn't always right: sometimes developers and users can see an obvious truth that the research community has completely overlooked!

When you look more closely at the CAP community, it is striking that they cite CAP in settings quite far from those Eric Brewer had in mind. As noted, the CAP *theorem* really is very narrow: it points to a tradeoff in a situation where transactions run against a highly available database split over two data centers. To get the result, the authors set up a case in which the system must guarantee availability for both replicas, even during periods when the WAN link connecting them is down, and then present the two subsystems with conflicting transactions: not a particularly general scenario. In contrast, the contemporary "CAP community" often takes CAP to mean that we should abandon consistency even in the first tier systems running in a single data center, when a network partition would either shut down the impacted computers, or the whole data center. For the theory community, this looks like a misapplication of CAP. But if pressed, the community that believes most strongly in CAP just simplifies it to CP: they tend to argue that consistency has a huge performance impact, and that in settings where scalable performance is the main goal, weakening consistency brings tremendous speedups. This sort of practical rule of thumb may not be what the CAP theorem expresses, but it does seem to be close to what Eric was thinking about, and it clearly has a strong following.

The irony (speaking as a person whose work is, in effect, challenged by both CATOCS and CAP), is that in my view, the issue isn't purely a practical one: I actually think that fundamental questions are present. The problem has been that CATOCS and CAP don't articulate those questions terribly well. Accordingly, what I hope to do in the remainder of this essay is to try and understand the sense in which our field has a set of core principles, and then to ask whether or not mechanisms that guarantee consistency defy those principles.

Yogi Berra, the baseball star, once remarked that "It's tough to make predictions, especially about the future." But sometimes the fix is in. I think this may be the case for us, too: the hardware platforms of the future are already in hand, even if research on leveraging that hardware is just getting underway. With this in mind, I'll end the essay with some speculation about what may come next, hoping that my guesses won't be too far off the mark.

**The Many Faces of Fault Tolerance**

The best place to start a review of fault tolerance research is by reiterating that the term can really be used in connection with all sorts of questions, some very different from the ones on which I'll focus here. The OS community has looked at fault tolerance at the level of the operating system itself (for example by making the OS tolerant of faulty device drivers and hardware [10][11][12]), and tolerating damage to persistent storage structures so that the OS won't crash if you try to mount a corrupted file system [13]). We've explored ways to mirror applications: Tandem's original *process pair* concept [14], for example, or the more ambitious perfect mirroring that Stratus Computer did back in the late 1980's with their "pair and a spare" hardware architecture [15]. We've become experts in bug-finding and code analysis, with entire conferences dedicated to such topics (and some great SOSP and OSDI papers, too[1]).

Beyond this our community has done a huge amount of work on logging, rollback, and checkpointing [18], and on using virtualization to contain failures or allow speculative executions [20][21]. We've done this at every granularity one could imagine: the process, sets of processes in a componentized application, the VM, sets of VMs and even in the file system. We've learned to mirror individual applications by logging events within a virtualized environment and even found ways to replay non-deterministic executions [19]. We know how to mirror file systems or entire data centers, and used those ideas to develop digital libraries that collaborate to preserve content[2] [22][23].

A significant area of study has focused on ways of either incorporating transactions directly into the OS or programming language environment [26][27] (an idea that went on to be widely adopted in industry [28]), or performing file system operations in ways that would leave the file system intact after a crash (and even here there have been many ideas: the early BSD file system work worried mostly about the OS interaction with file systems metadata [29], whereas systems like IBM's QuickSilver or CMU's Coda [30][31] went further and offered ways for applications to update sets of files atomically). For OS researchers from my era, late nights spent fixing a damaged file system after a crash was a price of developing new OS mechanisms; today, that sort of problem could never happen. Moreover, it has

[1] Here, I'll omit citations: I could make a list of some of the papers by Dawson Engler's group, or Yuanyuan Zhou's, but I'm not sure which to include. And this is will be the pattern: when I leave out citations entirely, you should just understand that I was lazy and didn't want to spend an hour to track them all down!

[2] Brewster Kahle, creator of the Internet Archive, once gave a talk at Cornell in which he remarked that the overarching goal of the Archive is to preserve humanity's collective knowledge even in the event of a global collapse of civilization. Hopefully, that particular goal won't ever be put to the test, but if a failure of that scale ever occurs, it would be nice to believe that we have the technology to recover from it!

become very easy to use transactions within modern programming environments such as the runtime support systems for Java, C# and C++ 11.

Another category of work starts by assuming a group of servers that might need to tolerate failures by some or all of the members. Here, we can talk about crash failures, network partitioning failures, or even data corruption and Byzantine behaviors, and each of those topics has an associated collection of seminal papers and results [32] , as well as more recent work that often takes an integrative approach [33][34][35], addressing several topics at once (for example, the Cambridge work on the CHERI processor is noteworthy for its breadth of coverage [24]). This recent trend is a departure from early work, which generally either tackles some specific category of failure (most work either looks at crash failures, or goes to the other extreme and assumes a Byzantine fault model), by the behavior desired (provably safe resumption of activity after a problem is resolved, versus continuous operation). There even is a whole school of thinking concerned with systems that can restore invariants even after a failure creates a period of havoc (the so-called *self-stabilization* methodology, proposed by Dijkstra [36] and then explored exhaustively by Shlomi Dolev and others [37]). On the other hand, the SOSP community hasn't really done much work with the self-stabilization approach. I've often wondered if we err by not doing so: it seems to me that self-stabilization theory might be useful as a rigorous way to formalize the BASE approach favored by the CAP community, a point to which we'll return shortly.

In the remainder of this essay and my talk I'll be focusing on consistent data replication of the kind used for fault tolerance in groups, for example in support of state machine replication (groups in which deterministic replicas are launched in the same state and then perform the same actions in the same order, hence go through the same evolution of states). The CATOCS and CAP conjectures are commonly perceived as criticisms of this class of systems, so we can drill down to understand the underlying tension that continues to trigger passionate debate decades after seemingly good solutions to the scientific problems were found.

## Consistency: Fault tolerance's Mirror Image

What does it actually mean to "tolerate a failure"? It's going to turn out that our story centers on how we answer this very basic question. Some of the *least* exciting work in the area stems from shallow, simplistic answers to the question: if you start with an extreme position it can be easy to prove that fault tolerance is absurdly costly, or impossible. But the resulting extreme claims have little to do with what OS engineers or large scale application designers are able to pull off. Systems builders are engineers, and don't need to guarantee that their systems will function even under the very worst scenarios imaginable. Our task is to build systems that should work well in the settings where they will actually be used, and that should tolerate reasonable patterns of failures. If the whole data center shuts down because power has gone out or the network has failed, well, obviously our fault tolerance solutions won't work until acceptable conditions are restored.

So with that in mind, let's lay out some basics. A good place to start is to realize that fault tolerance is inextricable from *consistency*, which is really just a formal way of talking about the correctness of a potentially complex system. In rough summary, we take some target system and create a list of properties it should satisfy. We call these *safety* properties: if they continue to hold, nothing bad will happen [38][39][40][41]. The collection of such properties defines the consistency of the application.

Notice that I haven't said anything about transactional ACID guarantees [42][43]. As readers of this essay will surely be aware, transactional databases offer a set of four core safety properties (atomicity, consistent operations that maintain data invariants, isolation and durability), so in this sense ACID is an instance of a consistency property. But one could equally well say that the Facebook caching infrastructure guarantees consistency provided that every image it returns to an edge client is one that was valid at some point in the past within the Facebook system [114]. This isn't an ACID guarantee and might violate "consistent caching," since some of those images be stale (think of a Facebook profile picture replaced by some newer profile picture for the same user). Yet if this is good enough for Facebook, perhaps we shouldn't think of it a weakening of consistency, but rather as a strong consistency statement that simply isn't identical to the ACID properties. Further, notice that Facebook's caching guarantees might not even be strictly weaker than ACID: perhaps Facebook achieves some form of probabilistic quality guarantee, or some kind of timeliness property. ACID does not cover such forms of consistency, so those properties would in some sense be stronger than what ACID offers!

In fact, Facebook would incur a meaningful "penalty" if the *typical* profile photo returned by its cache were stale, and it would completely unacceptable for it to substitute someone else's profile picture for mine. Amazon might have trouble selling products if their web pages constantly showed stale data: such a page might have incorrect pricing or could misreport the inventory for a highly desired product that has actually sold out. So one often brings additional safety properties into the mix. Notice that as I've expressed them, these are still safety properties, hence part of what I'll call the consistency model of the system. One could even write down a utility function describing the value to the system for enforcing such properties, and then build an infrastructure that maximizes utility. Such an approach would be inherently probabilistic.

In contrast, we use the term *liveness* when guaranteeing that something good will happen within bounded delay [38]. For example, we might want to guarantee that an Amazon inventory report on a web page is accurate to within 2.5 seconds. The difficulty is that as normally expressed by the theory community, liveness is an absolute, and hence can be a problem in a world where our goal is to engineer systems that simply need to be good enough to work well under normal operating conditions. All of those qualifications shift us away from the theoretical concept of liveness towards a more probabilistic form of conditional safety. Real systems aren't live, in the formal sense of the term.

In particular, Fischer, Lynch and Patterson (FLP) published a seminal paper in 1985 [44] , showing that in an asynchronous distributed system it is impossible to guarantee liveness for any protocol capable of solving a problem they formalized as *Fault-Tolerant Consensus* [48]. Consensus is simply the task of reaching agreement among a set of processes on an input: typically, 0 or 1. We insist that the value be an input to rule out trivial solutions (e.g. where everyone always decides 0). When we say that FLP holds in an asynchronous setting, we mean that no correct protocol can guarantee liveness if it will need to run in a system with no access to clocks and no bounds on message delay. The proof shows that if a protocol is correct, and is designed to tolerate crash failures (even just a single crash), there must be runs that take an unbounded number of steps and send unbounded numbers of messages, yet never reach a decision.

In some sense, FLP is a worst-case result: for example, network partitioning failures or data center power outages result in situations far more extreme than the one considered in FLP, and are more common too. With partitioning failures it is very easy to see that consensus cannot be achieved (suppose that you and I need to agree on a bit, and that the value must be a legitimate input and not a default value of some sort, but that the network is down and hence we cannot send messages to one-another. Game over: It can't be done). But FLP applies in a much more subtle situation, and as such, is a somewhat surprising result.

Yet while FLP establishes that we cannot guarantee liveness, it certainly doesn't prevent us from building protocols that achieve strong forms of consistency and that are also highly likely to tolerate the patterns of faults that real systems actually experience. There are other theory results that speak precisely to this question. A great example is the work of Chandra and Toueg on what they characterize as a "hierarchy" of failure detectors [45]. They show that if a failure detector satisfies a property at least as strong as a predicate they call <>W, then it can be used as the basis for a consensus protocol that will guarantee progress. The pragmatic implication is that as systems engineers, we can often make progress *extremely likely*. Moreover, <>W suggests a practical way to approach the question: if you want to build a fault-tolerant system, they recommend that failure detection be done by a failure detecting service, which should report failures and recoveries by some form of status update. This is exactly how modern data centers handle such issues: they have a “health monitoring” infrastructure.

I’ll just add one last digression before switching to focus on that question of deep principles. I find it fascinating that we have all sorts of results on Byzantine Fault Tolerance (BFT) for what are called synchronous networks: systems in which there are perfect clocks, and where the network runs in perfect rounds, and where every message sent is always delivered within the same round [46][47]. Clearly we can’t really build such hardware, yet we know more about BFT in this context than you could possibly imagine; there must be literally hundreds of papers, some very deep. These papers include actual code intended to run on a kind of computer will never exist.

Now, not all BFT work is aimed at code for imaginary, unrealizable computing devices. Here in the SOSP community we would tend to point to papers like the PRACTI work by Castro and Liskov [33], the so-called CASD (delta-T) atomic broadcast protocol of Cristian, Aghili, Strong and Dolev [49], or the Zyzzyva paper by Kotla and Alvisi [34], in which BFT is integrated with a gossip protocol [51] that can be used in real networks. But these results are not typical of the main body of BFT work, which really *is* focused on how to write code for an imaginary kind of data center that embodies unrealizable hardware.

So hold that thought, and now think back to what FLP tells us. A first thing to notice is that while the title of the FLP paper uses the word “impossibility”, in fact the justification for that particular choice of words will turn out to involve the formal definition of the term “impossible”, which the authors take to mean that there exists some way to attack the system that will prevent termination. To carry out an FLP-style attack on a real system, it is necessary to continuously tracking the state of every component and control the delivery time for every message sent.

When we put FLP next to BFT, we see a very strange juxtaposition. On the one hand, we have an impossibility result (a very unlikely one) for a kind of network we actually can build, and on the other, we have code that would solve BFT but only on a kind of computer that cannot be built! In an ideal world, we would want the exact opposite: a lot of work along the lines of PRACTI and Zyzzyva, or lots of solutions like the consensus protocol used in the <>W proof. But we have a surprising paucity of such work; the theory community seems to have a fascination with results inapplicable to practice.

They say that academic arguments are particularly nasty because so little is at stake, and perhaps this speaks to the point I want to make. As systems builders, our community has always been focused on pragmatics. We know perfectly well that most systems fail by crashing, and that mostly the failures won’t corrupt data. Real failures actually don’t result in bizarre worst-case Byzantine behaviors; for that, one has to imagine a virus compromising a few (f) servers, out of a larger set (at least 3f+1 would be needed), which can then overcome the failure. But 3f+1 is a very high price to pay for a very unlikely scenario. Anyhow, why wouldn’t the virus just infect the other 2f+1 machines?

So real systems builders are generally comfortable with 2f+1 replicas and frankly, in many cases we are perfectly happy with just f+1. After all, if one copy of the Internet Archive survives the global collapse of civilization, shouldn’t we just thank our lucky stars (and Brewster’s foresight), dust ourselves off, and rebuild? And this gets to the broader point: viewed from the theory perspective, systems builders seem frustratingly obtuse: Strangely comfortable building solutions that only “work” if one adopts the most tortured possible definition of what it means for a system to be correct. Worse, it really isn't easy to formalize a statement such as "my system will work as long as the data center is running, and my wide area system should keep the wide-area infrastructure consistent so long as the WAN link is working." Yet of course this is just what real systems builders do, all the time. The fact that in doing so we are flagrantly ignoring impossibility results seems not to trouble us.

Meanwhile, our counterparts on the theory side of the aisle have often had difficulty explaining why their proofs are even relevant to our real systems. Many in the systems community reject the models our theoretical counterparts favor, and once you reject the model, you certainly won't be very interested in the proofs. So all of us have had frustrating lunch discussions, in which our theory friends patiently explain that "Well, if our failures *could* be Byzantine, you'll need to employ a BFT model." We builders react by being baffled by their models, and appalled at the costs, and so we reject that friend's advice as unrealistic. Now things start a dangerous downward spiral: theory friend feels that this cavalier rejection of mathematics is seriously testing the friendship! Nobody wants to lose a friend, so the builder tries to explain that even though her or she isn't comfortable with the BFT model, or for that matter with the asynchronous model, it really would be awesome to prove the solution correct, and a bit of help would be welcome. But now theory friend repeats that this is hopeless, a waste of time: FLP makes it clear such an effort will fail. Why waste effort on a fatally flawed endeavor? Well, systems builder retorts (things are getting a bit emotional…), because if we don't convince ourselves that the solution works, it probably won't. Theory friend replies: "Well, it won't work." The systems person tries to salvage hope: "Hold on, that model you theory folks are using is kind of strange. Couldn't that give us an angle we could explore? Maybe *real* systems just aren't subject to these limitations?" "Well," theory friend responds (looking vexed), "if there is some other model that all of you systems folks are ready to endorse, pray tell, what precisely *is* that model?" This is perhaps a good moment to switch to a less controversial topic, like politics or religion.

Through countless such lunchtime dialogs, a schism has formed. For decades, the theory community has produced waves and waves of results that the systems community has completely ignored, even though some of those results turn out to be relevant to practice. Meanwhile the systems community has built huge edifices that the theory community has often ignored as incorrect by construction, even though in retrospect, one realizes that perhaps they are correct for the desired purpose, even if not correct relative to some very elegant but oversimplified model. I've wondered if we might not be missing all sorts of insights that could have been achieved had we found a way to work more constructively, rather than becoming lost in pointless arguments over models.

### Three Fundamental Systems Principles

With this context, it is time to briefly pivot and consider principles – the foundational beliefs of the systems community. In his history day talk, Peter Denning spoke of this, quoting Jim Gray on the idea that principles should be eternal: timeless, spaceless, and hopefully as relevant a hundred years from now as they are today. In fact, the evolution of hardware challenges many principles, so perhaps the hundred-year test is unrealistic. This said, the acronyms we've mentioned (BFT, FLP, CATOCS and CAP, ACID) certainly aspire to this kind of special status. Should we, in fact, recognize them as eternal principles of the systems community?

For our purposes here, I want to focus on a smaller set that seem especially relevant. A first principle was perhaps best expressed by Butler Lampson in his paper on Hints for Operating Systems Developers in 1983 [52]. Butler's paper covers a lot of ground (it is well worth reading even now, 32 years after it was written), but especially apropos was this: he argued that for most purposes, the OS developer faces the challenge of extracting some kind of core functionality that will be of broad value and power, then simplifying it as much as possible. Butler explains that our job is to take away everything that can somehow be removed from this core functionality, until what remains is as simplified and basic as possible. Then the next task is to focus on this remaining module, understand its critical performance-limiting paths, and to optimize those until the highest possible performance is achieved.

Notice how Butler's vision isn't purely about speed: he also values simplicity and structure, and treats these as three aspects of a single methodology. To me there is an elegance and clarity to this way of understanding systems: our task, if you accept this view, is to achieve a kind of balance between contending aspects of a single overarching process. In setting after setting, the systems community has encountered this tension, and I think Butler has it exactly right. These are three aspects of one methodology, inextricable and mutually interdependent. So here we have a principle that genuinely deserves the eternal, timeless, spaceless status Jim had in mind.

A second such principle generalizes the End-to-End Principle (E2E) of Saltzer, Reed and Clark [53]. As they originally stated it, the E2E principle was concerned with network functionality, and basically said that if the applications (the "end points" of a communication session) will need to implement some functionality no matter what the network layer does, then the network layer shouldn't duplicate that functionality unless somehow doing so brings a huge performance improvement.

The systems community seems to have generalized the E2E principle: our version echoes Butler's view that modules within the OS should be as basic and general as possible and shouldn't perform tasks that the application could perform instead, but then goes further to underscore that the OS should not impose a "model" on the higher level applications it supports. In effect, models, properties and guarantees must be viewed as application-layer abstractions. OS modules support abstractions but should themselves be general purpose building blocks that do not predetermine any behavior that the application-layer designer might prefer not to adopt: The OS should enable, not constrain.

So here we have two eternal principles. It will turn out that CATOCS and CAP are in some sense evocations of them, with CATOCS more focused on the generalized E2E principle, and CAP more focused on performance, as if the "P" in CAP was actually about performance, and not concerned with network partitioning at all. But in fact just as the OS community really transformed End-to-End in the process of embracing it, one could claim that the same has happened with CAP: those who cite CAP do so in a way that isn't precisely what Eric Brewer originally articulated. Indeed, as we will see shortly, this is true even for Paxos, the most famous of the strongly consistent replication solutions. What people refer to using the name Paxos isn't necessarily what Leslie Lamport had in mind when he coined the name. This tendency of our community is to seize upon things, but in doing so to also transform them, is striking. Perhaps this too is a deep and foundational principle: in systems, things often mean something other than what their creators initially had in mind. Ours is not a simple history, for better or for worse.

But before we tackle those complexities, there is actually one more foundational principle that I would like to cite, originating with Jim Gray. Jim was a good friend of mine, and a mentor. But he was also a mentor to the OS community, travelling long distances to join us at SOSP even though he published mostly in the database community, and forming strong personal ties to many of us, even though only some of those connections are reflected in papers. Jim had as much of a role in shaping the OS community as any of the other "principals" one could name.

Jim's contribution to our small list of core principles emerged from a paper he published at SOSP in 1985 where he looked at Tandem's "non-stop" computing systems and asked why the applications "stop" just the same [54]. His paper highlighted the role of bugs, specification errors, confused users, and similar effects. We've taken the resulting agenda to heart; today, we can point to dozens of papers on bug-finding, modular design techniques that compartmentalize problems and allow a system to survive the failure of some of its components, data replication in support of such behaviors, etc.

But the principle that I want to focus on was a bit different: in his 1985 talk, Jim directed some comments to the debate occurring at that time around the BFT model, commenting that if we really want to build fault-tolerant solutions, we would be wise to keep our eyes on the real needs of our real users. He made a similar remark in a talk he gave at an informal get-together hosted by IBM Almaden that we called "Byzantine Business" and was held around 1986, where he argued that if we really wanted to have impact on a key question, we should invest much more effort towards explaining the seemingly universal importance of the 2-phase commit protocol. (Many of the core BFT attendees were horrified… yet I think that the passing of time reveals that Jim was actually right: 2-phase commit has such a universal role, and this universality has never really been properly explained) To me Jim's comments mirror Butler's point about focusing on the critical path: Jim's additional insight was the recognition that the critical path that matters is the one that arises in real use. Such a comment can seem trite, but in fact this critical path is not necessarily one that might seem to be the most important one on the basis of a purely mathematical model, or even the one that seems to dominate in experiments performed using a test setup in the lab. For Jim, the real world was the ultimate testing ground and the needs of the real world were a source of inspiration.

Jim loved to use mathematics as a tool for reasoning about complex questions. So his point shouldn't be taken as anti-theoretical. Rather, he worried about our tendency towards an overly enthusiastic embrace of theories that often needed to be oversimplified to make them tractable. Throughout his career, Jim espoused this view, and his work was striking for his creative embrace of the complexity of real systems. For example one of my favorite papers tries to reduce the extremely complex world of replicated databases supporting ACID transactions to a set of very simple "rule of thumb" equations predicting scalability [55]. In that paper, Jim and his co-authors point out that unless you take care, every replica added to a database cluster will slow it down (his worst case scenario had an $O(n^5)$ slowdown!!). But Jim and his colleagues didn't stop with this finding and declare the "impossibility of database scalability." Instead, they turned the question on its head, treating the negative funding as a tool that led them towards other approaches that often worked incredibly well: sharding, for example, and partitioning database functionality in ways that reduce conflicts between concurrent transactions.

Oh, and with respect to the BFT point: Jim felt that real systems fail by crashing [54]. Others have since done studies reinforcing this view, or finding that even crash-failure solutions can sometimes defend against application corruption. One interesting study, reported during a SOSP WIPS session by Ben Reed (one of the co-developers of Zookeeper), found that at Yahoo, Zookeeper itself had never experienced Byzantine faults in a one-year period that they studied closely. But on the other hand, the clients of Zookeeper exhibited Byzantine behavior surprisingly often. This is actually doubly surprising, because we in the systems community have generally trusted the clients and, to the extent that we've studied BFT behaviors, we've tended to focus on servers running in replicated groups!

So let's summarize our three guiding principles, and then we can start to tackle fault tolerance and consistency in a crash-failure model.

I. Isolate core modules, simplify them as much as possible, and optimize the critical path.
II. Core modules should be general purpose, flexible, and fast. They should *enable* end-to-end properties and models but not *impose* decisions.

III. Focus on real-world systems, with real problems and real challenges. These may not match elegant models, but elegant models often steer you to overly complex, unscalable solutions.

Perhaps we should also pause just to note that at the end of the day, the systems community is performance-oriented. If solutions aren't fast, *really fast*, than the work is suspect. I mention this because this overarching fourth principle will turn out to be the most important of them all.

**Replication Techniques**

The 1983-1985 period also ushered in a series of practical solutions, including my own early work. To set some context, recall that in 1985 the SOSP community had seen a series of wonderful papers on transactional and database systems, and many people were debating ways that such mechanisms could be used in object-oriented runtime frameworks and systems. But the transactional model separates computation from storage, and many operating systems services and applications don't easily support this kind of separation: we also need to host applications where the data is simply in local variables within the processes making up the system, and where the platform can't somehow distinguish database state from application state. Further, the transactional model provides isolation between concurrently active tasks, but in many settings, the OS needs to support cooperation among concurrent actors.

This was one reason that Tommy Joseph and I became interested in the idea of replication as a fault tolerance tool for a more general class of arbitrary programs, with no associated database. In the 1985 period, there was much interest in state machine replication, an approach first suggested by Leslie Lamport in a 1974 paper [56] where he used a form of atomic broadcast to replicate a simple deterministic service. (Fred Schneider eventually expanded on the idea in a widely read tutorial [57]). In fact, as of 1985, there were no practical realizations of state machine replication, and if anything, the concept was perceived as being at odds with the trend towards multithreading (which brings inherent non-determinism) and the growing use of componentized software design (CORBA was in the air, although the OMG had not yet been founded). A componentized application might include a subsystem or two that use replication and yet the remainder of the application might not. In fact the applications sharing some object might not even be identical. This kind of reasoning led us to look at replication for complex systems implementing componentized abstractions. We concluded that the state machine replication model, which treated applications as static sets of service replicas, subsets of which might be unavailable, was a poor fit to the need. And while we loved its simplicity, we also concluded that the transactional model didn't fit the need.

The virtual synchrony model we came up with reflects this mix of what we knew from prior work and these new elements. A first paper [58], in 1985, showed how to accommodate a very dynamic style of replication in which processes could form groups, join and leave (or fail) dynamically, initialize themselves using a kind of checkpoint we called a state transfer, and use ordered multicast to update replicated state (we also treated membership of the group itself as a kind of replicated state, reporting it through "view-change" events). The system was also designed to handle network partitioning events by ensuring that if progress occurred, only the "majority partition" could advance; any minority partition would sense that it was disconnected from the main group and shut itself down. We proved that all of this would work correctly, and implement our solution in the first Isis system. But this first system was painfully slow.

In subsequent papers [59][60][61] we showed how Isis could be refactored into a membership service running a core membership protocol that we extracted from the 1985 system and called the group broadcast or Gbcast protocol, together with a set of protocols that consumed group views from this service and ran much faster by virtue of not duplicating the work already done in the membership service.

We also introduce a number of optimizations: ways to send multicasts asynchronously while preserving causal order, ways to deliver messages early (a form of optimistic or speculative execution), and then a primitive we called "flush" that would form a barrier, preventing an application from revealing these speculative states until they had become stable. Eventually, we refactored Isis into more of a library (we called it a "toolkit"), and in the process achieved speedups.

It wasn't long before the Isis Toolkit was running the New York Stock Exchange (it did so from around 1995 to roughly 2005, recovering from countless failures quickly enough so that no trading disruption ever occurred), the French air traffic control system (by now, Isis has a 20-year record without a single hiccup), the US Navy AEGIS warship, Oracle's network management system, a number of VLSI fab lines, telecommunications co-processors, and many other uses [62][63]. Quite a few of these Isis applications are still in use today.

But Isis wasn't the only practical distributed computing technology with fault tolerance properties. Earlier, I mentioned Jim Gray's Tandem study. Their process-pair technique for maintaining a hot standby was widely cited. At IBM, Flaviu Cristian developed a real-time mirroring technique that was ultimately applied in a project that sought to upgrade the US air traffic control system; the associated real-time multicast protocol was extremely interesting [49]: it starts as a simple all-to-all echo (so a multicast to N processes would trigger $N^2$ messages), but then adds additional mechanisms to overcome various failure patterns and features to order messages and ensure that all deliveries occur within a short window of time. On the other hand, this protocol struggles when one tries to run it rapidly, leading to some very curious failure modes, as discussed in [63]. Leslie Lamport introduced his first paper on Paxos in 1989 [16]. We saw a sudden surge in interest in publish-subscribe communications, including a SOSP paper on the TIB system [64]. All of this set the context for the 1993 iteration of SOSP, when CATOCS suddenly dominated hallway conversations [65][66][67].

### The Origins and Evolution of the Paxos Protocol

But before tackling CATOCS, I want to pause to say a few words about Paxos, because that work has turned out to be so impactful. It is important to realize that the Paxos protocol existed before Leslie's Part Time Parliament paper [16][69]. That paper was first circulated in Technical Report form late in 1989, and it wasn't published in ACM TOCS until 1996. Leslie picked a cute name for the protocol, and it stuck, but the protocol itself predated his focus upon it, a point that Keith Marzullo discusses in a sidebar that was published with the TOCS paper when it appeared in print [16][17]. Much of this section echoes the contents of that sidebar, although I can add a bit more color and a few details Keith omitted.

Exactly when did Paxos originate? For those of us working in the field at the time, Paxos didn't really spring upon us fully formed, but rather emerged from a series of incremental steps. Among these was the work on Isis just mentioned: the Gbcast protocol used in that 1985 version of Isis [58] and then "refactored" to become the main protocol in the Isis membership oracle in the 1987 and subsequent versions [59][60][61] was actually solving a Paxos-like problem. It turns out that Gbcast bisimulates Paxos [68][69], in the sense that any run that can occur in the classic Paxos protocol (the one Lamport focused on) can also occur in Gbcast, and any run possible for Gbcast is also possible for Paxos[3]. Thus the existence of Gbcast had some of us thinking about other kinds of protocols that could run in groups with other properties: ordering, freedom from partitioning oddities ("split brain" syndrome), and strong enough to make the state of an application durable even across complete shutdowns. The Isis Toolkit had

[3] In fact, [68] includes an informal recipe for transforming Paxos into Gbcast, or vice-versa.

a number of such protocols, typically derived from Gbcast but with relaxations of one or another of its properties, and intended for use by applications that didn't actually need those particular properties.

But knowing that Gbcast can bisimulate Paxos doesn't make Gbcast the first of the Paxos protocols. In fact, under any sensible similarity metric, Gbcast and Paxos are very different. In particular, Gbcast has some properties that Paxos lacks: for example, Gbcast allows a client to send streams of asynchronous requests and will preserve the request ordering. Paxos does not allow this form of asynchronous streaming: in the classic Paxos protocol, the client awaits responses and so can only send an asynchronous stream by forking off threads: each request would then be treated separately, and hence requests could be reordered. Of course, a client could batch requests and send several as one Paxos operation, or include some form of event ordering directly into the request, which the application could then enforce.

The above may sound trivial, but it really isn't. Consider the following. There is an failure case in which Paxos decides to leave a gap in the message order: some leader proposes X, and another leader proposes Y, and another leader proposes Z. None is successful. Now suppose that the leaders for Y and Z re-propose their commands in some other slot, but the leader proposing X fails, and hence X never commits. Paxos can end up with a gap in its command sequence. Paxos includes logic for sensing and ignoring such a gap.

But the effect is that if a client intended that X, Y and Z be done in sequence, we ended up with Y and Z committed in the Paxos log, but X is stuck in an uncommitted state: perhaps present in a number of acceptor logs, yet lacking a leader and essentially abandoned. This leaves Y and Z orphaned in the eyes of the application: learnable (since they committed), but not "applicable" in the sense that the application intended that X be performed first. Dahlia Malkhi has pointed out that in an even more obscure scenario, the group of acceptors might later be reconfigured to add a new acceptor (for example because some other acceptor has suffered an irreparable hardware fault that damaged the disk). In such a situation, it would be common to initialize the new machine by copying the log from some existing acceptor to the joining acceptor (a discussion of this can be found in Chapter 22 of [63]). Notice however, that in doing so, we replicate a log in which X may be present: recall that it was proposed for that original slot. Thus, by creating one additional copy of this log, X could actually become committed long after it was initially proposed, and long after the system moved onward leaving a gap for that slot in the Paxos log! Of course such issues are solvable; Malkhi has a nice solution, and Gbcast (which guarantees gap freedom) had a solution to this issue from the outset.

There are also some differences in the other direction: Paxos has what might be called an implied notion of durability across full failures followed by recoveries: in such cases, the Paxos definitions require that the ***protocol*** state stored in the acceptor logs be learnable (Paxos doesn't have an explicit definition of durability, per-se). In contrast, Gbcast treats durability as a property of the ***application*** state, providing tools to help the application make its state durable. Thus if a Paxos service is restarted from a total failure, Paxos itself remembers its past history; if a Gbcast-based service is similarly shut down and restarted. Gbcast remembers nothing. The application is responsible for remembering its own state. Gbcast per-se is stateless. Perhaps the most important difference is that the Gbcast proof was very clumsy compared to the Paxos proof. Given the different treatment of causal gaps and durability, the fact that Gbcast looks very different from Paxos, and the awkward proof structure, it is clear that Gbcast wasn't the first Paxos.

Turning to the history summarized in Keith Marzullo's sidebar, which today can be updated with additional discussions of prior work (notably in papers by a colleague of mine, Robbert van Renesse [76]), one can construct a loose history of other relevant prior work. The timeline I've arrived at starts in

1978 [56], when Leslie Lamport discussed replicated state machines [57] for the first time, but didn't talk about asynchronous environments or how to solve consensus in such environments. This of course was long before any of the events mentioned above.

By 1982, it was widely believed that consensus in an asynchronous environment with crash faults was impossible, but FLP offered the first strong proof [44], and the authors began to talk about their work around this time. While not published until 1985, FLP was thus known to hold.

In 1983, Michael Ben-Or designed working consensus protocols for an asynchronous environment with faults [71]. This protocol responded to FLP by proving termination with probability 1, technically a weaker guarantee that the form of total correctness defined in the FLP work. The Ben-Or protocol achieves consensus with 2f+1 participants (matching the lower bound), and there is also a Byzantine version that uses 5f+1 participants (this exceeds the lower bound). This paper appeared as an extended abstract in PODC without proof, but even so, introduces rounds (similar to Paxos ballots) and quorums. On the other hand, Ben-Or doesn't have a similar notion of leaders.

Also in 1983, Bracha and Toueg, inspired by Ben-Or, proved lower bounds for both crash and Byzantine faults and presented protocols that meet the lower bounds and terminate with probability 1 (assuming a network that delivers messages in random order—an assumption Ben-Or did not make, and a bit unconvincing for those of us who work with real networks). This work appeared as a full paper within the same PODC [72]. Again, the protocol lacked the Paxos concept of leaders.

In 1984 Dwork, Lynch, and Stockmeyer designed and proved correct the first leader-based consensus protocol [73]. To address FLP, this paper adds an assumption (Global Stabilization Time) under which the protocol can be proven to terminate, and their protocol is thus totally correct with this assumption. However, while this protocol has similar power to Paxos, in much the same sense that Gbcast is similar in power to Paxos, it would be very hard to make a case that the protocol "is" Paxos.

We've already discussed my work in the 1985-1987 period, so I won't repeat that.

In 1988, Oki and Liskov published a paper on a protocol they called Viewstamped Replication (VR) [70]. The important innovations include the following:

- VR supports consensus applied to a primary-backup style replicated database.
- Like Paxos, VR is a "multi-decree" protocol: it is not necessary to solve consensus from scratch for each decision, and they offer this as an advantage, arguing that VR has overhead similar to that of a standard primary-backup approach.
- VR has "round skipping": participants don't have to wait for a round to finish before moving on to the next. The feature is useful if multiple leaders are faulty, and Paxos has a similar mechanism.

Based on these structural similarities to Paxos, many consider VR to be the first true Paxos protocol, but it is not trivial to simply accept this assertion. The problem is that as presented, VR is not a clean Paxos protocol: The paper presents the protocol in the context of a database replication system built using it, and the "state" of the protocol is in fact the replicated database itself. The original paper only includes a proof sketch, very informal, but the actual proofs, which did not appear until much later in Oki's thesis, focus on the correctness of this composed system (protocol+database) and as a result, are not at all similar to the Paxos proofs in structure or style. This is very important, because the Paxos proof structure is what made Lamport's work so notable.

Thus, when Lamport first began to circulate his Part-Time Parliament paper in 1989 (it was submitted to ACM TOCS in 1990), the work was striking not so much because it included the Paxos protocol, but also because the formal treatment of the protocol established a framework that has turned out to be of great importance to the subsequent evolution of work in the area. The paper per-se is famously difficult to read, starting with its frequent use of people's names rendered phonetically into Greek. Setting that issue to the side, within it one finds a protocol similar to Viewstamped Replication, but applied to State Machine Replication, and associated with careful descriptions of the system model and proofs of safety. No liveness argument is given. Thus while the paper may not be to everyone's taste, one cannot dispute that the Paxos protocol is described in a careful and ultimately, beautiful, way.

I would argue that Lamport's very elegant proof by refinements, in which a basic property is established and then preserved while layering in a new property and a new proof, was tremendously innovative and a departure from any of the prior work I've mentioned. A series of papers were written to try and simplify the exposition of the proof: these include one by de Prisco [74], one by Lamport [75], and one by Van Renesse [76]. I highly recommend this last [76]; for me, it is the most readable of all. But it the more obscure failure cases that the Paxos proof addresses remain fairly hard to understand: the ideas are simple. The complexity arises because it is hard to visualize all the possible states of a distributed system in the face of failures or transient network partitioning events. Indeed, Lamport himself worried that it would be very hard to convince oneself that implementations of Paxos were correct, and went on to create a whole formalism for constructing machine-checkable proofs using a notation he calls TLA+, which can then be model checked [77]. In adopting model checking, Lamport actually uses an approach to Paxos that is widely associated with Lynch [46]. A recent contribution in this spirit is the proof by Hawblitzel [95].

In other work, Bickford, Rahli, Van Renesse and Constable proved Paxos using a constructive logic (one in which expressions can be true, false, or undecidable, and hence proof by contradiction is not possible). They start by defining a domain-specific language called the *logic of events* [79][80][82], then express Paxos in this logic, and then are able to synthesize a formally correct implementation of the protocol from the proof, and even a provably correct database recovery algorithm [81]. They also have a constructive derivation of FLP [83]. They also have used this technique to manipulate virtual synchrony protocols, for example to synthesize versions optimized for specific patterns of use [89].

This idea of starting with one protocol and transforming it into a family specialized in various ways is universal within the field. For example, after introducing the original Paxos protocol, Lamport also went on to create a whole family of Paxos variants, each retaining the core Paxos guarantees (non-triviality, request ordering and durability), but specialized for particular settings: Paxos with an elected leader (this shifts it closer to Gbcast), Paxos with membership dynamics (this was done very differently than in Isis), Paxos with batching (needed for speed), Disk Paxos (for cases in which the storage system played the acceptor role, but lacked "intelligence"), etc. Some don't even bisimulate classic Paxos [78].

Who, then, invented Paxos? I'm not sure the question can be answered, although Brian Oki and Barbara Liskov clearly deserve recognition for writing the protocol down as part of their database research. On the other hand, it is interesting to realize that Viewstamped Replication and Paxos do not actually bisimulate, a point Van Renesse touches upon in [78], concluding that Viewstamped Replication and Paxos both refine (backward simulate) yet another, higher-level protocol. Both Viewstamped Replication and Paxos do use a notion of ballot, leader, round-skipping, multi-decree, and quorums, but this work argues that one can't map executions of one onto executions of the other, let alone in both directions at once. They do assume the same failure model and provide the same form of state "durability", but in Viewstamped Replication, the protocol logs are actually used directly as the state of the replicated database itself.

But the invention of Paxos turns out not to have ended even after Lamport gave the protocol that name. Over time, the name Paxos came to be attached to protocols that don't bisimulate Paxos. Lamport himself has done so, although he only uses the Paxos name for protocols strong enough to solve consensus, and with some rigorously defined notion of protocol state, non-triviality, ordering and durability. Chubby, the widely cited Google lock manager [84][85], is described by its authors as having been based on Paxos, but they also describe an extensive series of optimizations and extensions they made. We tend to cite Chubby as the first highly visible real-world use of Paxos, but the actual Chubby protocol really isn't one you would find in Lamport's papers. The same occurred when Zookeeper [90] was created: the developers (Flaviu Junquera and Ben Reed) describe themselves as having initially thought about basing Zookeeper on Paxos, but then making optimization after optimization. By the time they finished this process, Zookeeper was remote from Paxos and if anything, more reminiscent of the virtual synchrony protocols used in Isis – although again, not identical. Meanwhile, in fact, the Isis implementation of virtual synchrony was itself evolving, away from the 1987 version into a more complex "out of band" control structure that resulted in higher performance but brought new complexity into the system [61].

All of this created a degree of confusion. At one point, the Isis virtual synchrony model (the model implemented by Gbcast) was commonly described as being in competition with state machine replication and Paxos. But this was just a skin-deep impression: At the 2009 "Replication: Theory and Practice" workshop at Monte Veritas (the "Summit of Truth") in Lugano, Dahlia Malkhi and Leslie Lamport described a model that incorporated elements of the virtual synchrony model together with elements of the classic Paxos model, which permitted them to correctly support asynchronous event streaming and to leverage the virtual synchrony notion of a dynamic group of processes (they called this a stoppable and reconfigurable state machine [91] , but referred to the model as "Virtually Synchronous Paxos"). In follow-on work, Dahlia, Robbert van Renesse and myself extended the idea to create a formal structure in which a true Paxos protocol can live side by side with the optimistic style of protocols used in the Isis Toolkit; this is discussed in Chapter 22 of my book [63], and is implemented in the Vsync system[4] . The library has been downloaded some 5250 times.

To summarize, the technology we associate with Paxos emerged around 1988, acquired the Paxos name in 1990, and became increasingly successful during the late 1990's, eventually displacing most other options. However, this summary conceals a deeper and much less linear story: the real story involves many versions of Paxos (widely different protocols), unified in their ability to solve consensus-like problems and to provide totally ordered updates to replicated data, but rather different if compared carefully. They often look very different, and some don't bisimulate classic Paxos. These more mature versions of Paxos commonly use leader election to avoid contention for slots in the Paxos command list, batch large numbers of commands per Paxos operation, support multiple leaders with ownership of disjoint Paxos slots, and may also include application-specific optimizations. At least one uses dynamic group membership tracing back to the early Isis approach. Viewed this way, Paxos is more of a category of protocols than any single, specific protocol. Ironically, and confusingly, it has come to include even work that was done before the classic Paxos paper had been published.

Which is the *purest* version of Paxos? To my taste, the recent work on Corfu [87] and Tango (sometimes referred to as Corfu-DB) [88] yields the service truest to the real Paxos definition. Paxos is best understood as a system that maintains an append-only log on durable storage, offering a way to truncate

[4] Vsync was initially named Isis2, but the name was changed to avoid accidental association with the ISIL terrorist organization. The download count in the text includes downloads from the older site: Isis2.codeplex.com.

log prefixes if desired, but otherwise offering precisely the Corfu API.    Yet this simple statement might surprise some in the field!

I'm always surprised that when I say explain to my students that Paxos defines a totally ordered and durable message log, many respond that no, Paxos is actually a kind of atomic multicast solution.  Those who think of Paxos as an atomic multicast are (in my view, but also Leslie Lamport's), making a natural but incorrect association.  There is no question that Paxos resembles atomic multicast, just as virtually synchronous atomic multicast resembles Paxos.  Nonetheless, because Paxos holds data – has a notion of durable state – it turns out to be notoriously hard to use in applications that have their own ***distinct*** notion of durable state.  Recall the work by Oki and Liskov in which Paxos was actually the data storage subsystem of their Viewstamped Replication database solution [70]: This is really how Paxos was conceived by Lamport as well.  In contrast, with reliable multicast, it makes sense to talk about a layer that sits in front of the replicated database: a relaying stage that presents each update to each replica, but then lets the replicas maintain the true database state (a reliable multicast has no durable state of its own).

The puzzle that arises if we put Paxos into this sort of staged configuration is that if the application has some sort of state external to Paxos, then it needs a synchronization mechanism to coordinate management of its own durable state with management of the durable state stored in Paxos itself.  In some ways, this explains why the Viewstamped Replication protocol is hard to separate into a Paxos-like protocol composed with a distinct, free-standing database application: with Paxos, it is actually more natural to just build a database in which the protocol state "is" the application state, exactly as Viewstamped Replication approaches the matter.  But if one wants to take a database like Oracle or SQL server and treat it as a black box, and then put Paxos in front of it to replicate updates, the durability issue leaps to the forefront and turns out to be very challenging – this is precisely where the Isis notion of providing tools to help the developer enters the picture.  Paxos lacks such tools.

In fact, to use Paxos as a replication layer composed with a stateful application (in the sense of "applications with their own state, stored next to each of the replicas"), the application itself would generally need to be idempotent, meaning that if the same events are presented more than once, the application would be responsible for discovering this, filtering duplicates, and applying each event exactly once and in the proper order.  This is because when Paxos restarts after a complete failure, a component "relearns" the full contents of the durable log, and the best intuitive interpretation of the "learn" mechanism is to think of it as a protocol that presents a sequence of events to the application replicas, which each are expected to process that full sequence. It is up to the replicas of the application itself to understand which commands within the log have already been processed by the local instance: they must distinguish commands that have been applied to the application state from those that still need to be applied.  Further, the Paxos formalism doesn't explain the conditions that the client must establish before it is safe to request log truncation, or offer a formal meaning for reconfiguration from the perspective of external application state.

I should end this section by noting that the family of solutions we've discussed are not the only options in this space.  For example, when using replication for fault tolerance, many systems prefer to just implement some form of checkpoint-rollback [92][93], or adopt the chain-replication approach [94] (a generalized form of "warm standby" with f+1 replicas: updates are applied to the head of the chain and relayed to the tail; reads are done on the tail, where state is f+1 fault tolerant).  These are simple techniques, well understood, and they have been shown to work really well even at huge scale.  Further, they are supported by all sorts of software platforms, so you typically don't need to roll your own.  They have connections to state machine replication, but the techniques don't require remotely as much heavy machinery as for a dynamically managed process group with multiple leaders each proposing batches of commands against distinct slots and with heavy levels of asynchrony.

Thus, the past three decades have left us with a family of protocols that has many members: all the variations of Paxos, the virtual synchrony systems (Isis Toolkit, Isis2, Transis, Ensemble, Horus, JGroups), Chubby, Corfu, Zookeeper, Sinfonia, chain replication, Raft, Zab, etc. Members of the family can all solve consensus: the most fundamental problem. Some treat failure detection and membership management as a core service, some batch operations for speed, some offer asynchronous streaming, some leverage optimistic early delivery with a flush barrier, some exploit a form of leader election, and this is just a few of their many features. To a surprising extent, these protocols are actually different versions of a single story; sometimes (as with Gbcast and classic Paxos) we can find a series of edits that would basically transform one protocol into another; in other cases there are extra features or other changes that prevent such a transformation.

Personally, I think of *durability* as an open question and a bit of a puzzle: many systems offer some way to make an application durable, but whereas some focus on the durability of the protocol itself, others treat durability as a question of application state, and still others solutions just don't offer durability, but are intended for settings like publish-subscribe, where that property typically isn't needed. Above I discussed one aspect of the issue, but there is another aspect that may be of growing importance soon: the rapid emergence of a wide variety of non-volatile memory options, making it increasingly feasible to persist just about anything we like, without the huge overheads seen in traditional rotational file systems. High persistence delays had more of an influence on shaping protocols like Paxos than many might realize, and while one can definitely create a system like Corfu that optimizes Paxos for an NVRAM log, the broader question we need to ask concerns what durability should *mean* in applications that are replicated for various reasons. Not every application will want to keep its state in a Corfu log. To me, this highlights a sense in which durability could easily emerge as one of the dominant questions of the next era in distributed systems research.

At any rate, all members of this consensus family share core correctness guarantees, and stretching over the entire family, we have a proof methodology popularized by Lamport (but with roots in prior work), permitting a powerful style of incremental proof by refinement, and supports model-checking of the code implementing the protocol. Today, we can honestly say that these solutions are good ones, widely used by the industry, and with the performance required to support even very demanding use cases.

### The CATOCS controversy

So with this backdrop, let's return to CATOCS. The authors assert that:

- Systems of this sort violate the generalized End-to-End principle: they impose a model on their users. They claim that these systems are too complex to drop into the kernel and that they lack a clean decomposition compatible with Butler Lampson's vision of a kernel composed of clean, simple building blocks that can be heavily optimized and then used in flexible, general ways.
- Causal and total ordering are just two from a spectrum of possible ordering guarantees that could also include real-time features, special features to let applications leverage commutativity or other properties of the workload, etc. Each application will need its own special mix; causal or total order will invariably prove to be too much, or too little.
- Ordering guarantees are costly and by imposing them, we deny the user a chance to pay (only) for properties actually needed. The CATOCS paper goes further, and asserts that causal order is complex to support and representation of causality potentially involves a data structure that will grow as $O(n^2)$,

where n is the number of processes using the system. Total order, CATOCS argues, is even worse, requiring contention for an ordering lock or token.

- They sum all of this up with an argument that at the end of the day, CATOCS solutions are simply too slow: much slower than what might be theoretically feasible on the same hardware. In effect, they claim that if application developers understood the cost, they would always reject strong ordering.

How valid were these claims? It is certainly true that early versions of Isis and Paxos were fairly slow and relatively complex, so one should grant this point, at least with respect to those implementations. But over time, such systems became faster and faster and modern implementations are extremely performant. Some, like the highly modular Horus [96][97], Transis [98][99], Ensemble [100][101] and JGroups [102] systems, or the Libpaxos version of Paxos [103], are spare and elegant, with building block components that could be dropped into the kernel (although I am not aware that anyone has actually explored such a kernel-hosted option). So this particular claim really depends on which version one is looking at, and even at the time was dubious. It certainly isn't a universal criticism of all such solutions.

One can also dispute the CATOCS assertions about the cost of ordering. Even by 1993 papers had been written on how to compactly represent ordering information [61][62][67][104][105]. Moreover, the CATOCS $O(n^2)$ claim makes sense only for causal order tracking for point to point messages. This is relevant because in Isis, which is where that issue arose, causal ordering of this kind was available only for group multicasts, where an O(n) vector can cover the worst case needs, and a compressed one might be much smaller. Further, Isis only used this feature in support of a particular form of lock-based concurrency control [106]. Better is to view these systems as using a gamut of tools for generating order, tracking order and enforcing it. It strikes me as an overreach to brush those to the side and assert that all of them are "too complex" or to claim that all "scale poorly."

CATOCS also asserts that there will be as many kinds of ordering needs as there are applications, but do different applications really need different forms of order? The example given in the paper involved real-time [1], and they do have a point: many applications do have real-time considerations: a dimension not considered in the papers that CATOCS was focused upon. On the other hand, one can add this property: Google's Spanner system [107] merges real-time with logical consistency, and Spanner is just one of many such solutions (I like this example because Spanner scales remarkably well).

The hardest question and perhaps the strongest of the many CATOCS points is perhaps the "slower than the hardware, and in opposition to our deepest principles" assertion. For example, CATOCS views protocols such as the ones we mentioned as being overly complex. There is no question that systems strong enough to solve consensus and able to tolerate failures are complicated, but on the other hand, the modular systems cited earlier are quite elegant. And turning the question on its head, the implementation of a file system with strong security can be complicated too. A thing should be as simple as possible, but not simpler (at least not if you want it to work correctly). So to argue that these mechanisms are too complex is really to argue that users don't need them in any case. Yet there is ample evidence that that consistency and total ordering are needed in real systems. Our obligation, then, should be to offer it in a simple, elegant and highly optimized form, not to reject it out of hand.

But it remains true that we haven't done a very good job of isolating simple, robust building blocks that look elegant and easily fit within the OS kernel. Today, the OS kernel component most relevant to data replication or reliable multicast is IP multicast, but this protocol is astonishingly unreliable and can actually destabilize entire data centers [108]. The other option is TCP: not a great match for replication.

But perhaps the world is shifting under our feet in ways that broaden the notion of what it means to be an "OS module." The tools we've talked about exist mostly at the user layer, as libraries or stand-alone services, and in the past, might have been classified as "middleware." But the introduction of solutions like U-Net [109] and PCAP [110] was game-changing. Today, as we look towards growing use of RDMA and NVRAM, the kernel plays a diminishing role in moving bytes or persisting them because the hardware is so much faster than anything we can do in software.

I'll leave it to the reader to reach their own verdict on the CATOCS controversy. But as noted in the introduction, the 13 or so papers in the 2015 edition of SOSP that would seem to fall into this category of systems is perhaps the final word on CATOCS. Total order, causality and strong forms of consistency have clearly found a place in our community.

**CAP Conjecture**

Meanwhile, a relatively recent rebirth of CATOCS has occurred under a different acronym. When Eric Brewer first began his work on aggressively updated caching systems for the cloud (soft-state replication), he was onto something most of us had overlooked: the majority of today's cloud applications can manage perfectly well with potentially stale cached data, provided that the application developer is prepared to do a little extra work to mask the oddities that can occur when reading cached information that might no longer be correct. CAP reflects the recognition that for queries to give snappy responses in Europe, or Asia, there isn't much of a choice: if they can't respond using locally cached data, the delay of waiting to interrogate a backend-database that might be 125ms distant n California can be prohibitive. Thus CAP relaxes cache consistency to gain better availability and partition tolerance. The CAP *theorem* formalizes this for a consistent (ACID) database confronted with conflicting updates during a period when a WAN link has broken, partitioning it into two disconnected subsystems [9][111].

Earlier we noted that CAP has been generalized by the cloud community, where a curious dichotomy can be perceived: the cloud infrastructure developers themselves, a sophisticated community, often use methods with *strong* consistency guarantees, like Spanner. That is, they don't necessarily impose CAP on their own solutions. In contrast, the APIs offered to PaaS application developers provide significantly weaker guarantees. Thus a web page rendered by a typical app running on the Apple cloud, or Azure, or Google's App Engine will often be served cached images and other content that might be stale, with no indication that the content isn't the most current version. The designer is expected to minimize the extent to which the end-user would become aware of this. If inconsistency does occur and somehow is evident within the system itself, a background reconciliation occurs [112]. This gives rise to a methodology known as BASE [113]: BASE rejects the database ACID guarantees and instead offers a system that is basically available (BA), uses soft state (S) for most end-user interactions, then then employs a background mechanism to achieve eventual consistency (E). For example, a background program might check a database against various data invariants, notice situations that violate the invariants, and then apply some sort of correction that will restore the invariant but might not be exactly what the external user would have expected. Infrastructure owners who promote BASE frequently cite CAP, and then explain that in their environment, strong consistency is at odds with high performance, especially at scale. Network partitioning is viewed as a secondary concern. And yet as noted, those same infrastructure owners often use stronger consistency in their own infrastructure subsystems!

BASE is often touted as a rejection of consistency but it is interesting to ask just how weak this model actually turns out to be. After all, distributed systems don't work correctly for magical reasons. Rather, there is a process of mutual accommodation: the designer adjusts the specification and the implementation until they match and also correspond to a highly scalable solution. In this sense, CAP and BASE are simply justifications for an accommodation that promotes higher performance at cloud scale.  Further, notice that even a cache that might contain stale data still has a meaningful guarantee: the cached data should at least reflect a real state that system was once in.   Further, BASE systems don't exactly revel in inconsistency: most try very hard to *minimize* inconsistency, recognizing that there is a negative utility associated with disappointing the end-user, and one of the 2015 SOSP papers documents this for Facebooks caching infrastructure, showing that even with CAP, they actually achieve very high consistency most of the time [114].  A Christmas shopper who sent her grandchildren a train set and the matching tracks would not be pleased if they opened the box to find only tracks.  On the other hand, that Facebook study didn't consider situations when Facebook was experiencing failures or under some other form of rare stress, so there could be times when CAP temporarily permits significant inconsistencies.

And what of the basic concern, namely that stronger forms of consistency are just too slow?  I haven't had time here to discuss transactions and ACID guarantees, but as noted earlier, Google Spanner would seem to be an existential proof that global scale transactional solutions really can be made to work.  Yet whereas systems like the various Paxos libraries, or Corfu, or my Isis2 platform are pretty fast, particularly if requests are batched and the server receiving them is multithreaded, they are fairly slow relative to what the hardware could potentially be capable of if running at full speed.

With modern RDMA hardware carrying the bytes, multicore platforms, and NVRAM or CrossPoint or similar non-volatile storage, we are able to move data at speeds of 100Gb/s today, with 200, 400 and a terabit expected soon, and making that data non-volatile simply requires selecting blocks of memory that live on a non-volatile storage technology.  In recent work in my lab, we built a reliable multicast over RDMA that runs at the full speed of the optical backplane when making as many as 32 replicas, and is still hitting nearly half the optical speed when replicating data to 512 nodes, and the HPC community uses a version of the MPI library that achieves similar speed on Infiniband.  Contrast this with what we might call good performance today for systems like Paxos or Isis2: in most papers, data rates in the 100's of Mb/s are considered pretty reasonable, using tricks like batching request and employing different leaders for different Paxos slots.  Thus there is a missing factor of perhaps 100x, and that gap might be more like 10,000x without the kinds of tricks just mentioned, which basically squeeze lots of work into each action.

Durability is a mystery too: as noted earlier, Paxos makes its *own* state durable, which is not necessarily what an application using Paxos is concerned about.  The underlying design decisions often recall an era of rotating disks, where file access latencies were many thousands of times slower than DRAM access delays, and hence there was a significant risk of inconsistency if data was delivered to an application before the disk write had completed.  Thus if today we can make memory persistent just by requesting that a region be mapped to non-volatile storage, and the only impact is an access latency of a few nanoseconds for each block of data, we need to ask the obvious question:  Does it even make sense for these protocols to be structured as they presently are?  I suspect not.

### Discussion: CATOCS and CAPS in Context

What then are we to conclude about CATOCS and CAP?  As we've seen, there are many ways to rebut both principles.  Neither rises to Jim Gray's test of timeless, spaceless and eternal truth.  Yet in picking them apart and arguing that CATOCS gets the overhead costs of ordering wrong, or that the C and the P in CAP are often interpreted very differently than in the CAP theorem, we risk losing track of a deeper truth: both CATOCS and CAP speak to a valid concern.

This concern, for me, centers on the three legitimate principles cited early in the essay.  I really believe that core systems components should be modular, powerful, general, and not impose decisions on the applications that use them.  I also agree with Jim that the most important problems to solve are the ones that arise in real systems.  As we survey the landscape of fault tolerance and consistency in October of 2015, it seems clear that we haven't done remotely enough to show that our solutions really rise to the standards articulated by these foundational truths.

Consistency mechanisms remain complex, and difficult to use. Worse, to make them fast we often refactor them into separate modules but those modules remain so tightly coupled that the effect isn't to simplify the solution.  A membership layer or a leadership election protocol that runs in a separate system component doesn't qualify as separate module if there are countless calls and callbacks between it and the multicast protocol or message log code that uses it, depending upon the precise behavior of the membership or election mechanism, and doing so in subtle ways.  We lose the clean separation of concerns that Butler urged us to strive for.

Further, when one structures protocols in this way, it turns out that many of the resulting modules have functionality duplicated in other standard OS components, yet protocol designers rarely tamper with the OS.  As a result, because we leave those other elements untouched, the OS as a whole is left with redundant and inconsistent handlings of the corresponding functionality.  For example, it is common for a multicast protocol or a message log to have a specialized notion of failure detection.  But failure detection also arises in the TCP protocol layer, and in the file system.  Thus if we leave them to implement their own, totally separate (and inconsistent) approach to detecting failures, the OS as a whole now has three or four ways of sensing failures.  When we modularized Paxos or Isis we didn't really modularize failure handling in the OS as a whole.  We simply ended up with anarchy: localized structure, global chaos.  A coherent treatment of failure handling would have created a single module for that purpose, and then would have modified the entire operating system to use it in a consistent manner.  Doing *that* would abstract the notion of failure.  What we have done up in systems like Paxos and Isis is purely local to those elements and in this sense, ignores Butler's wise advice.

As it turns out, there are many such interdependencies.  We discussed durability in Paxos and the puzzle this has started to pose.  But just as failure sensing occurs throughout the OS, so do durability decisions.  Today, we seem to treat NVRAM and other persistent memory stores as file systems, which certainly simplifies naming and block management, but might not there be other, cleaner ways to manage this property better matched to uses like creating a persistent log?  Memory management itself is a puzzle too: I haven't discussed the point here, but when doing RDMA transfers from memory, the memory pages from which data is sent or received need to be preallocated.  Often, for security reasons, the OS will also zero such pages if this is the first time the application has had access to them.  They will need to be pinned to prevent paging or swapping, and preregistered in the virtualized IO-bus memory map.  In some systems the easiest option is to scatter-gather received data but then, once the actual received length is known, map the received blocks into a continuous segment using VM remapping because applications run fastest on contiguous memory regions.  And then there is the need to translate from the addressing the application is using into the addresses the NIC should use for DMA transfers.  Thus communication protocols that run at the very fastest speeds must be deeply integrated with the memory management abstractions of the OS.  And even here there are additional such questions: for example, multicore issues.  In a NUMA architecture, the pages used for communication really should be local to the cores that will later be running the application code likely to touch the data.  At RDMA speeds we need to pin threads just to poll for I/O completions.  All of these observations lead towards a radically different notion of how an OS with a deep embrace of data replication should be structured.   That approach would respond to Butler's advice, and to the generalized E2E principle; the approach we take now is piecemeal even if the protocols per-se are more and more modular and elegant.

To me, the deep truth of CATOCS and CAP is this: while we've solved state machine replication at moderate scale and with reasonably good success, and have many impressive real-world uses of the resulting solutions that we can cite to back up the claim, there is a fundamental sense in which we've yet to actually tackle the contemporary version of the consistency question. Needed is a more coherent OS structure in which the key elements relevant to data replication and failure handling are modularized and decomposed cleanly into separate components with clearly distinguishable roles, able to interoperate cleanly and to give us the various flavors of Paxos, but perhaps also used to support memory management in the rest of the OS, or durability for application-managed objects of other kinds. Security overheads such as the zeroing of memory should be taken off the critical path. Failure detection should be a service shared by all components and protocols that need to tolerate failures. Then these modules will need to be assembled into a more familiar form, but in ways that promote clarity of function and elegance of the overall edifice, and that don't introduce excessive cross-module dependencies. In contrast, today we may have a reasonably elegant way to reason about Paxos, but are very far from a coherent view of this full structure, and might even need to rethink Paxos as part of the undertaking, because the assumptions used by Paxos may turn out to in some ways be at odds with the goals of such a redesigned OS structure.

The overarching questions centers on performance: the most important metric of all. We aren't remotely near the achievable performance for modern infrastructures, as the experiments I mentioned earlier underscore. The authors of the CATOCS and CAP work, at the end of the day, should probably just have pointed out that systems like Isis and Paxos were slow and left it at that; the resulting criticism would have been much harder to rebut. Moreover, a performance-based observation would have been reasonably timeless: as we just noted, even modern Isis2 or Paxos implementations move data 100x (if not 10,000x) slower than the optical backplane data rate, and with latencies that are correspondingly poor compared to what might theoretically be feasible. One can certainly graft such systems to RDMA and NVRAM, and they will run faster, but this presupposes that the old choices of primitives are the best fit for the new architectures. As we have seen, I very much doubt that this is the case.

And so to conclude, what fascinates me about the historical CATOCS and CAP debate is that while none of the participants back in 1993 did a particularly good job of articulating their positions (myself included), there does seem to be an enduring validity to at least one important aspect of the CATOCS and CAP concerns. Taken as shallow statements, one can demolish either proposition. In a similar sense, CAP and CATOCS themselves took the goal of consistency in a shallow way, and demolished the resulting mischaracterization of the property. But if we agree not to be petty, we need to acknowledge that there is a real point here, and that this real point is well taken. The consistency mechanisms we've been working with, and refining and polishing, do work well. But they are surprisingly expensive, not terribly modular, fit poorly into the overall OS, and impose architectural decisions that increasingly seem to be at odds with modern hardware and system goals!

### Disruption Is Coming

Not long in the future our community will help create a new generation of cloud systems with strong consistency needs: smart cars, for example, will need help from smart cloud transportation services. Those, in turn, will depend on cloud-scale solutions that track sensors to monitor conditions throughout smart highways and cities. Smart homes will be connected to a smart power grid that optimizes power generation and consumption to minimize the need for environmentally damaging coal-burning power stations. Smart medical systems will offer health advice night and day. Air traffic control systems will propose special routings to save fuel or time, and will dynamically replan flight schedules to ensure that preferred customers won't miss connections. The airlines themselves may shift towards just-in-time seating. The coming world will be a true Internet of Things, it will be smart, and all of this will depend upon a new generation of cloud computing systems. Today's dumb systems may be fine with CAP and BASE, but tomorrow's smart ones will need more.

In this light, the realization that consistent replication seems to be 100x or more slower than should be possible on today's RDMA and NVRAM platforms might not be a damning criticism. On the contrary, such an observation begins to look like a tremendous opportunity. We have a great reason to reinvent consistency and fault tolerance in a completely new setting, and the hardware has shifted in ways that also offer us all sorts of new architectural options. In future systems, data will move out of band (I'm thinking of the control plane / data plane separation promoted in systems like BarrelFish [115], Araxis [116] and iX [117]), memory will be as durable as we chose for it to be, and multicore parallelism and cloud-scale parallelism will prevail. SOSP and OSDI are impressed by 10x performance improvements, so the students who tackle such problems will be able to publish at least a few cycles of papers on their work. Moreover, the hardware is so different that there appears to be a real chance for completely new ideas.

When he read this essay, Peter Neumann shared a few comments but then concluded with a powerful observation: he quoted a conversation with Butler Lampson in which Butler remarked that resilience without security is pointless, and that security without resilience is inadequate. Peter's point was that in focusing on fault tolerance and consistency without discussing security, I've discussed just one side of a multifaceted (well, at least two-faceted) question. His point is very well taken, and it is absolutely true that a security perspective could be brought to bear on almost every topic I've raised. Indeed, the logic formalisms used to reason about security and those used to reason about consistency are in a deep sense the same ones: a security system that behaves inconsistently would be useless, and in a similar sense, a consistent system incapable of supporting secure tasks such as credentials management is probably not powerful enough to play other roles that require strong guarantees. Certainly, I wouldn't want to be driven around by a smart car dependent on an insecure cloud computing infrastructure. However, this essay is already far too long, so I'll leave the topic as an exercise to the reader: I am sure that study of the ramifications of security in the context of the coming generation of systems will reveal all sorts of opportunities to build amazing systems and to write wonderful papers about them!

And with that thought, I'll conclude. The next steps will need to be taken by the young members of our community. In fact, you'll have competition: my group is certainly going to work on these topics too. But I suspect that at the next SOSP history day, it will be my turn to attend, while someone else explains all the surprises that the coming 20 or 25 years of progress on fault tolerance will bring!

**Acknowledgements**

This essay benefitted enormously from discussions with Peter Denning and Peter Neumann, Robbert van Renesse, Lorenzo Alvisi, Fernando Pedone and Bob Constable, beyond which I could list a dozen more who heard rehearsals of my talk and suggested valuable changes. I also want to express gratitude to the agencies that have recently funded my work, notably DARPA, NSF and DOE.

**Cited Materials**

[1] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. Proc 14th SOSP (1993). http://dx.doi.org/10.1145/168619.168623

[2] Robbert van Renesse. Causal controversy at Mont St.-Michel. *SIGOPS Oper. Syst. Rev.* 27, 2 (Apr 1993), 44-53. http://dx.doi.org/10.1145/155848.155857

[3] Ken Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *SIGOPS Oper. Syst. Rev.* 28, 1 (Jan 1994), 11-21. http://dx.doi.org/10.1145/164853.164858

[4] Robbert van Renesse. Why bother with CATOCS?. *SIGOPS Oper. Syst. Rev.* 28, 1 (Jan 1994), 22-27. http://dx.doi.org/10.1145/164853.164859

[5] Eric Brewer. Towards Robust. Distributed Systems. Dr. Eric A. Brewer. July 19, 2000. https://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[6] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. Proc 18th SOSP *2001*. http://dx.doi.org/10.1145/502034.502057

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. Proc *21st* SOSP, 2007. http://dx.doi.org/10.1145/1294261.1294281

[8] Dan Pritchett. BASE: An Acid Alternative. *Queue* 6, 3 (May 2008), 48-55. http://dx.doi.org/10.1145/1394127.1394128

[9] Seth Gilbert and Nancy Lynch. Perspectives on the CAP Theorem. *Computer* 45, 2 (Feb 2012), 30-36. DOI=http://dx.doi.org/10.1109/MC.2011.389

[10] Peter J. Denning. Fault-Tolerant Operating Systems. ACM Comput. Surv. 8, 4 (December 1976), 359-389. DOI=http://dx.doi.org/10.1145/356678.356680

[11] 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. Proc 15th SOSP (1995), 267-283. http://dx.doi.org/10.1145/224056.224077

[12] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. Proc. 22nd SOSP (2009), 59-72. http://dx.doi.org/10.1145/1629575.1629582

[13] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Membrane: Operating system support for restartable file systems. Proc. 8th USENIX conference on File and storage technologies (FAST ’10), 2010. http://dx.doi.org/10.1145/1837915.1837919

[14] Joel F. Bartlett. 1981. A NonStop kernel. Proc 8th SOSP (1981). ACM, New York, NY, USA, 22-29. http://dx.doi.org/10.1145/800216.806587

[15] Michel Banatre and Peter A. Lee. Hardware and Software Architectures for Fault Tolerance. Springer Verlag Lecture Notes in Computer Science, 774. February 28, 1994

[16] Leslie Lamport. The Part-Time Parliament. ACM Trans. Comput. Syst. 16, 2 (May 1998), 133-169. http://dx.doi.org/10.1145/279227.279229. First circulated as a technical report in 1989.

[17] Keith Marzullo. Sidebar published with The Part-Time Parliament. ACM Trans. Comput. Syst. 16, 2 (May 1998), 134 and 160. http://dx.doi.org/10.1145/279227.279229.

[18] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34, 3 (September 2002), 375-408. http://dx.doi.org/10.1145/568522.568525

[19] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: efficient online multiprocessor replay via speculation and external determinism. In Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS XV). ACM, New York, NY, USA, 77-90. DOI=http://dx.doi.org/10.1145/1736020.1736031

[20] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP '05). ACM, New York, NY, USA, 191-205. DOI=http://dx.doi.org/10.1145/1095810.1095829

[21] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. SIGOPS Oper. Syst. Rev. 36, SI (December 2002), 211-224. http://dx.doi.org/10.1145/844128.844148

[22] Hakim Weatherspoon, Lakshmi Ganesh, Tudor Marian, Mahesh Balakrishnan, and Ken Birman. Smoke and mirrors: Reflecting Files at a Geographically Remote Location Without Loss of Performance. In Proceedings of the 7th conference on File and storage technologies (FAST '09), Margo Seltzer and Ric Wheeler (Eds.). USENIX Association, Berkeley, CA, USA, 211-224.

[23] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. SIGARCH Comput. Archit. News 28, 5 (November 2000), 190-201. http://dx.doi.org/10.1145/378995.379239

[24] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization, IEEE Symposium on Security and Privacy, San Jose, CA, May 18-20, 2015.

[25] Butler Lampson. Redundancy and Robustness in Memory Protection, Information Processing 74, Proceedings of the IFIP Congress 1974, North-Holland, Amsterdam, Hardware II, pages 128--132, 1974.

[26] Barbara Liskov. Distributed programming in Argus. Commun. ACM 31, 3 (March 1988), 300-312. DOI=http://dx.doi.org/10.1145/42392.42399

[27] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In Proceedings of the 1996 ACM SIGMOD international conference on Management of data (SIGMOD '96), Jennifer Widom (Ed.). ACM, New York, NY, USA, 318-329. DOI=http://dx.doi.org/10.1145/233269.233346

[28] Microsoft Language Integrated Query Domain Specific Language. https://en.wikipedia.org/wiki/Language_Integrated_Query

[29] M. McKusick, W. Joy, S. Leffler, R. Fabry, ``A Fast File System for UNIX", ACM Transactions on Computer Systems 2, 3. pp 181-197, August 1984.

[30] Frank Schmuck and Jim Wylie. Experience with transactions in QuickSilver. In Proceedings of the thirteenth ACM symposium on Operating systems principles (SOSP '91). ACM, New York, NY, USA, 239-253. DOI=http://dx.doi.org/10.1145/121132.121171

[31] Satyanarayanan, M. "The Evolution of Coda" ACM Transactions on Computer Systems, Volume 20, Number 2, May, 2002

[32] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. 4, 3 (July 1982), 382-401. DOI=http://dx.doi.org/10.1145/357172.35717

[33] Miguel Castro and Barbara Liskov. 2002. Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. 20, 4 (November 2002), 398-461. http://dx.doi.org/10.1145/571637.571640

[34] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine fault tolerance. ACM Trans. Comput. Syst. 27, 4, Article 7 (January 2010), 39 pages. http://dx.doi.org/10.1145/1658357.1658358

[35] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The next 700 BFT protocols. In Proceedings of the 5th European conference on Computer systems (EuroSys '10). ACM, New York, NY, USA, 363-376. DOI=http://dx.doi.org/10.1145/1755913.1755950

[36] Edsger W. Dijkstra. 1974. Self-stabilizing systems in spite of distributed control. Commun. ACM 17, 11 (November 1974), 643-644. DOI=http://dx.doi.org/10.1145/361179.361202

[37] Shlomi Dolev. 2000. Self-Stabilization. MIT Press, Cambridge, MA, USA.

[38] E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. Commun. ACM 8, 9 (September 1965), 569-. DOI=http://dx.doi.org/10.1145/365559.365617

[39] P. Brinch Hansen. Concurrent programming concepts. Computing Surveys 5 (1973), 223-245.

[40] Leslie Lamport. 1986. The mutual exclusion problem: part I—a theory of interprocess communication. J. ACM 33, 2 (April 1986), 313-326. DOI=http://dx.doi.org/10.1145/5383.5384

[41] Leslie Lamport. 1986. The mutual exclusion problem: partII—statement and solutions. J. ACM 33, 2 (April 1986), 327-348. DOI=http://dx.doi.org/10.1145/5383.5385

[42] Gray, Jim. "The Transaction Concept: Virtues and Limitations" (PDF). Proceedings of the 7th International Conference on Very Large Databases. Cupertino, CA: Tandem Computers. pp. 144–154. Sept. 1981.

[43] Haerder, T.; Reuter, A. "Principles of transaction-oriented database recovery". ACM Computing Surveys 15 (4): 287. 1983. doi:10.1145/289.291.

[44] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. J. ACM 32, 2 (April 1985), 374-382. DOI=http://dx.doi.org/10.1145/3149.21412.

[45] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. J. ACM 43, 2 (March 1996), 225-267. DOI=http://dx.doi.org/10.1145/226643.226647.

[46] Nancy Lynch. Distributed Algorithms. Morgan Kaufman, 1996.

[47] Danny Dolev, Ruediger Reischuk, H. Raymond Strong. Early stopping in Byzantine agreement. September 1990 Journal of the ACM (JACM): Volume 37 Issue 4, Oct. 1990

[48] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. 1987. On the minimal synchronism needed for distributed consensus. J. ACM 34, 1 (January 1987), 77-97. DOI=http://dx.doi.org/10.1145/7531.7533

[49] Flaviu Cristian, Houtan Aghili, Ray Strong, Danny Dolev. Atomic broadcast: from simple message diffusion to Byzantine agreement. Information and Computation 118 (1): 158–179, 1995.

[50] Danny Dolev, Dalia Malki. The Transis approach to high availability cluster communication. March 1996. Communications of the ACM: Volume 39 Issue 4, April 1996.

[51] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic algorithms for replicated database maintenance. In Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC '87), Fred B. Schneider (Ed.). ACM, New York, NY, USA, 1-12. DOI=http://dx.doi.org/10.1145/41840.41841.

[52] B. Lampson. Hints for computer system design, Proceedings of the Ninth ACM Symposium on Operating Systems Principles (Bretton Woods, New Hampshire, United States) 1983, pages 33--48.

[53] J.H. Saltzer, D.P. Reed, D.D. Clark. End-to-end arguments in system design,. ACM Transactions on Computer Systems Volume 2, Issue 4 (November 1984), pages 277--288.

[54] Jim Gray. Why Do Computers Stop and What Can Be Done About It? SOSP, 1985.

[55] Jim Gray, Pat Helland, Patrick O'Neil, Dennis Shasha. The Dangers of Replication and a Solution. ACM SIGMOD 1996.

[56] Leslie Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems, ACM TOPLAS 6:2, 1974.

[57] Fred Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial, ACM Computing Surveys Volume 22, Issue 4 (December 1990), 299--319.

[58] Ken Birman. Replication and Fault-Tolerance in the ISIS System. 10th ACM Symposium on Operating Systems Principles, Dec 1985, 79-86.

[59] Ken Birman and Thomas. A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. 11th ACM Symposium on Operating Systems Principles, Dec 1987.

[60] Ken Birman, Thomas. A. Joseph. Reliable communication in presence of failures. ACM Transactions on Computer Systems, Vol. 5, No. 1, Feb. 1987.

[61] Ken Birman, Andre Schiper and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. ACM Transactions on Computer Systems, Aug 1991, (3):272-314.

[62] Ken Birman and Robbert van Renesse. Reliable Distributed Computing with the Isis Toolkit., IEEE Computer Society Press, 1994.

[63] Ken Birman. Guide to Reliable Distributed Systems. Springer Verlag, 2010.

[64] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. 1993. The Information Bus: an architecture for extensible distributed systems. In Proceedings of the fourteenth ACM symposium on Operating systems principles (SOSP '93). ACM, New York, NY, USA, 58-68. DOI=http://dx.doi.org/10.1145/168619.168624

[65] David R. Cheriton and Dale Skeen. 1993. Understanding the limitations of causally and totally ordered communication. In Proceedings of the fourteenth ACM symposium on Operating systems principles (SOSP '93). ACM, New York, NY, USA, 44-57. DOI=http://dx.doi.org/10.1145/168619.168623

[66] Robbert van Renesse. 1994. Why bother with CATOCS?. *SIGOPS Oper. Syst. Rev.* 28, 1 (January 1994), 22-27. DOI=http://dx.doi.org/10.1145/164853.164859

[67] Ken Birman. 1994. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. SIGOPS Oper. Syst. Rev. 28, 1 (January 1994), 11-21. DOI=http://dx.doi.org/10.1145/164853.164858

[68] Gbcast Protocol. https://en.wikipedia.org/wiki/Gbcast

[69] Paxos Protocol. https://en.wikipedia.org/wiki/Paxos_(computer_science)

[70] Brian Oki, Barbara Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of Distributed Computing. pp. 8–17. doi:10.1145/62546.62549.

[71] Michael Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. Proceedings of the Second ACM Symposium on Principles of Distributed Computing, pages 27-30, August 1983

[72] Gabriel Bracha and Sam Toueg. 1983. Resilient consensus protocols. In Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC '83). ACM, New York, NY, USA, 12-26. DOI=http://dx.doi.org/10.1145/800221.806706

[73] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. J. ACM 35, 2 (April 1988), 288-323. DOI=http://dx.doi.org/10.1145/42282.42283

[74] Roberto de Prisco, Butler Lampson, and Nancy Lynch. Revisiting the Paxos Algorithm. Theoretical Computer Science 243, 1-2 [46](July 2000), pp 35-91

[75] Lamport, Leslie (2001). Paxos Made Simple ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) 51-58.

[76] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. ACM Comput. Surv. 47, 3, Article 42 (February 2015), 36 pages. DOI=http://dx.doi.org/10.1145/2673577

[77] Leslie Lamport. The TLA+ Home Page. http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html

[78] Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider. Vive la Difference: Paxos vs. Viewstamped Replication vs. Zab. IEEE Transactions on Dependable and Secure Computing. Vol. 12. No. 4. July 2015.

[79] Bickford, Mark and Constable, Robert L. Formal Logical Methods for System Security and Correctness, in Formal Foundations of Computer Security 14: 2008 (29-52).

[80] Mark Bickford, Robert L. Constable, Vincent Rahli. The Logic of Events, a framework to reason about distributed systems. 2012, http://www.nuprl.org/documents/Bickford/LOE-LADA2012.html

[81] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford and Robert L. Constable. Developing Correctly Replicated Databases Using Formal Tools. DSN 2014, 395—406.

[82] Rahli, Vincent and Bickford, Mark and Anand, Abhishek. Formal Program Optimization in Nuprl Using Computational Equivalence and Partial Types. In Proceedings of the 4th International Conference on Interactive Theorem Proving (ITP 13), 261-278, Springer-Verlag.

[83] Robert L. Constable. Effectively Nonblocking Consensus Procedures Can Execute Forever: a Constructive Version of FLP. Cornell University, 2008, Tech Report 11512.}

[84] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, Nov, 2006.

[85] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live – An Engineering Perspective. PODC '07: 26th ACM Symposium on Principles of Distributed Computing, 2007.

[86] The Vsync Cloud Computing Library. https://en.wikipedia.org/wiki/Vsync_(computing). Download site: http://vsync.codeplex.com.

[87] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A distributed shared log. ACM Trans. Comput. Syst. 31, 4, Article 10 (December 2013), 24 pages. DOI=http://dx.doi.org/10.1145/2535930

[88] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: distributed data structures over a shared log. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13). ACM, New York, NY, USA, 325-340. DOI=http://dx.doi.org/10.1145/2517349.2522732

[89] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. 1999. Building reliable, high-performance communication systems from components. In Proceedings of the seventeenth ACM symposium on Operating systems principles (SOSP '99). ACM, New York, NY, USA, 80-92. DOI=http://dx.doi.org/10.1145/319151.319157

[90] Flavio P. Junqueira and Benjamin C. Reed. 2009. The life and times of a zookeeper. In Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC '09). ACM, New York, NY, USA, 4-4. DOI=http://dx.doi.org/10.1145/1582716.1582721

[91] Leslie Lamport, Dahlia Malkhi and Lidong Zhou. Reconfiguring a State Machine. ACM SIGACT News Volume 41, Issue 1 (March 2010).

[92] Lorenzo Alvisi and Keith Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. IEEE Trans on Software Engineering, 24:2, Feb 1998, 149-159.

[93] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34, 3 (September 2002), 375-408. DOI=http://dx.doi.org/10.1145/568522.568525

[94] Robbert van Renesse and Fred B. Schneider. 2004. Chain replication for supporting high throughput and availability. In Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04), Vol. 6. USENIX Association, Berkeley, CA, USA, 7-7.

[95] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. ACM Symposium on Operating Systems Principles (SOSP 25), Nov 2015, Monterey, CA.

[96] Robbert van Renesse, Kenneth P. Birman and Silvano Maffeis, Horus, a flexible Group Communication System, Communications of the ACM, April 1996.

[97] Robbert van Renesse, Masking the Overhead of Protocol Layering ACM SIGCOMM Conference, Stanford, September 1996.

[98] Danny Dolev and Dahlia Malkhi. Proceedings of the Dagstuhl Intl. Workshop on Unifying Theory and Practice in Distributed Computing, (LNCS, 938), Dagstuhl Castle, September 1994.

[99] Danny Dolev , Dalia Malki. The Transis Approach to High Availability Cluster Communication. Communications of the ACM, April 1996.

[100] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building Adaptive Systems Using Ensemble. Software--Practice and Experience. Vol28, No. 9, pp, 963—979, August 1998.

[101] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Ken Birman, and Robert Constable. Building Reliable, High-Performance Communication Systems from Components. In Proc. of the 17th ACM Symposium on Operating System Principles, Kiawah Island Resort, SC, December 1999.

[102] Bela Ban. JavaGroups: Support for Group Communication Patterns in Java. Technical report, available from http://www.jgroups.org/papers.html

[103] Daniele Sciascia, Fernando Pedone. Libpaxos. https://bitbucket.org/sciascid/libpaxos

[104] Colin J. Fidge (February 1988). "Timestamps in Message-Passing Systems That Preserve the Partial Ordering" (PDF). In K. Raymond (Ed.). Proc. of the 11th Australian Computer Science Conference (ACSC'88). pp. 56–66. Retrieved 2009-02-13.

[105] Mattern, F. (October 1988), "Virtual Time and Global States of Distributed Systems", in Cosnard, M., Proc. Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France: Elsevier, pp. 215–226.

[106] Thomas Joseph and K.en Birman. Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems. ACM Transactions on Computer Systems 4, 1 Feb.1986.

[107] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. Proceedings of OSDI'12: Tenth Symposium on Operating System Design and Implementation, Hollywood, CA, October, 2012.

[108] Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, Robert Burgess, Haoyuan Li, Gregory Chockler, Yoav Tock. Dr. Multicast: Rx for Data Center Communication Scalability. Eurosys, April 2010 (Paris, France). ACM SIGOPS 2010, pp. 349-362.

[109] Thorsten von Eicken, Anindya Basu, Vineet Buch and Werner Vogels. .U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, December 1995.

[110] https://en.wikipedia.org/wiki/Pcap.

[111] Seth Gilbert and Nancy Lynch, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”, ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59.

[112] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07). ACM, New York, NY, USA, 205-220. DOI=http://dx.doi.org/10.1145/1294261.1294281.

[113] Dan Pritchett. BASE: An ACID Alternative. ACM Queue 6:3, July 28, 2008.

[114] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, Wyatt Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. In Proceedings of twenty-fifth ACM SIGOPS symposium on Operating systems principles (SOSP '15). ACM, Monterey CA.

[115] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In 22nd ACM Symposium on Operating Systems Principles (SOSP), Big Sky, MT, USA, October 2009.

[116] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. ACM Transactions on Computer Systems (TOCS), 33(4), November 2015

[117] Adam Belay, George Prekas; Ana Klimovic, Samuel Grossman, hristos Kozyrakis, Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. OSDI, 2014 (Broomfield, CO).