Chapter 6
A History of the Virtual Synchrony Replication Model

Ken Birman

Abstract In this chapter, we discuss a widely used fault-tolerant data replication model called *virtual synchrony*. The model responds to two kinds of needs. First, there is the practical question of how best to embed replication into distributed systems. Virtual synchrony defines dynamic *process groups* that have self-managed membership. Applications can join or leave groups at will: a process group is almost like a replicated variable that lives in the network. The second need relates to performance. Although state machine replication is relatively easy to understand, protocols that implement state machine replication in the standard manner are too slow to be useful in demanding settings, and are hard to deploy in very large data centers of the sort seen in today's cloud-computing environments. Virtual synchrony implementations, in contrast, are able to deliver updates at the same data rates (and with the same low latencies) as IP multicast: the fast (but unreliable) Internet multicast protocol, often supported directly by hardware. The trick that makes it possible to achieve these very high levels of performance is to hide overheads by piggybacking extra information on regular messages that carry updates. The virtual synchrony replication model has been very widely adopted, and was used in everything from air traffic control and stock market systems to data center management platforms marketed by companies like IBM and Microsoft. Moreover, in recent years, state machine protocols such as those used in support of Paxos have begun to include elements of the virtual synchrony model, such as self-managed and very dynamic membership. Our exploration of the model takes the form of a history. We start by exploring the background, and then follow evolution of the model over time.

6.1 Introduction

A "Cloud Computing" revolution is underway, supported by massive data centers that often contain thousands (if not hundreds of thousands) of servers. In such systems, scalability is the mantra and this, in turn, compels application developers to replicate various forms of information. By replicating the data needed to handle client requests, many services can be spread over a cluster to exploit parallelism.

B. Charron-Bost, F. Pedone, and A. Schiper (Eds.): Replication, LNCS 5959, pp. 91–120, 2010.

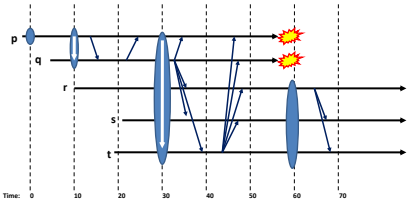

Fig. 6.1 Synchronous run.

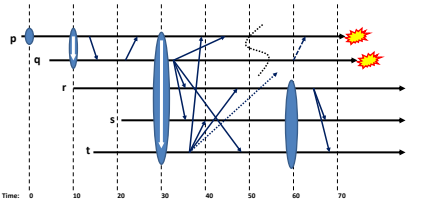

Fig. 6.2 Virtually synchronous run.

Servers also use replication to implement high availability and fault-tolerance mechanisms, ensure low latency, implement caching, and provide distributed management and control. On the other hand, replication is hard to implement, hence developers typically turn to standard replication solutions, packaged as sharable libraries.

Virtual synchrony, the technology on which this article will focus, was created by the author and his colleagues in the early 1980's to support these sorts of applications, and was the first widely adopted solution in the area. Viewed purely as a model, virtual synchrony defines rules for replicating data or a service that will behave in a manner indistinguishable from the behavior of some non-replicated reference system running on a single non-faulty node. The model is defined in the standard asynchronous network model for crash failures. This turns out to be ideal for the uses listed above.

The Isis Toolkit, which implemented virtual synchrony and was released to the public in 1987, quickly became popular [40, 14, 65, 10]. In part this was because the virtual synchrony model made it easy for developers to use replication in their applications, and in part it reflected the surprisingly good performance of the Isis protocols. For example, Isis could do replicated virtually synchronous updates at almost the same speed as one could send raw, unreliable, UDP multicast messages: a level of performance many would have assumed to be out of reach for systems providing strong guarantees. At its peak Isis was used in all sorts of critical settings (we'll talk about a few later). The virtual synchrony model was ultimately adopted by at least a dozen other systems and standardized as part of the CORBA fault-tolerance architecture.

Before delving into the history of the area and the implementation details and tradeoffs that arise, it may be useful to summarize the key features of the approach. Figures 6.1 and 6.2 illustrate the model using time-space diagrams. Let's focus initially on Figure 6.1, which shows a nearly synchronous execution; we'll talk about Figure 6.2 in a moment. First, notation. Time advances from left to right, and we see timelines for processes p, q, r, s and t: active applications hosted in a network (some might run on the same machine, but probably each is on a machine by itself). Notice the shaded oval: the virtual synchrony model is focused on the creation, management and use of *process groups*. In the figures, process p creates a process group, which is subsequently joined by process q, and then by r, s and t. Eventually p and q are suspected of having crashed, and at time 60 the group adjusts itself to drop them. Multicasts are denoted by arrows from process to process: for example, at

time 32, process q sends a multicast to the group, which is delivered to p, r, s and t: the current members during that period of the execution.

Process groups are a powerful tool for the developer. They can have names, much like files, and this allows them to be treated like topics in a publish-subscribe system. Indeed, the Isis "news" service was the first widely used publish-subscribe solution [8]. One thinks of a process group as a kind of object (abstract data type), and the processes that join the group as importing a replica of that object.

Virtual synchrony standardizes the handling of group membership: the system tracks group members, and informs members each time the membership changes, an event called a *view change*. In Figure 6.1, new group views are reported at time 0, 10, 30 and 60. All members are guaranteed to see the same view contents, which includes the ranking of members, the event that triggered the new view, and an indication of whether the view is a "primary" one, in a sense we'll define just below. Moreover, virtually synchronous groups can't suffer "split brain" problems. We'll say more about this topic later, but the guarantee is as follows: even if p and q didn't actually fail at time 60, but simply lost connectivity to the network, we can be sure that they don't have some divergent opinion about group membership.

When a new member joins a group, it will often need to learn the current state of the group — the current values of data replicated within it. This is supported through *a state transfer*: when installing a new view that adds one or more members to a group, the platform executes an upcall in some existing member (say, q) to request a state checkpoint for the group. This checkpoint is then sent to the joining member or members, which initialize their group replica from it. Notice that state transfer can be thought of as an instantaneous event: even if a multicast is initiated concurrently with a membership change, a platform implementing virtual synchrony must serialize the events so that the membership change seems atomic and the multicast occurs in a well-defined view.

The next important property of the model concerns support for group multicast. Subject to permissions, any process can multicast to any group, without knowing its current membership (indeed, without even being a member). Multicast events are ordered with respect to one-another and also with respect to group view events, and this ensures that a multicast will be delivered to the "correct" set of receivers. Every process sees the same events in the same order, and hence can maintain a consistent perspective on the data managed by the group.

Now, consider Figure 6.2. We referred to the run shown in Figure 6.1 as *nearly synchronous*: basically, one event happens at a time. *Virtual synchrony* (Figure 6.2) guarantees an execution that looks synchronous to users, but event orderings sometimes deviate from synchrony in situations where the processes in the system won't notice. These departures from synchrony are in situations where two or more events commute. For example, perhaps the platform has a way to know that delivering event a followed by b leaves q in the same state as if b was delivered first, and a subsequently. In such situations the implementation might take advantage of the extra freedom (the relaxed ordering) to gain higher performance.

We mentioned that protocols implementing virtual synchrony can achieve high update and group membership event rates — at the time this chapter was written, in

2009, one could certainly implement virtual synchrony protocols that could reach hundreds of thousands of events per second in individual groups, using standard commodity hardware typical of cloud computing platforms.[1] We'll say more about performance and scale later, but it should be obvious that these rates can support some very demanding uses.

In summary: virtual synchrony is a distributed execution model that guarantees a very strong notion of consistency. Applications can create and join groups (potentially, large numbers of them), associate information with groups (the state transferred during a join), send multicasts to groups (without knowing the current membership), and will see the same events in equivalent orders, permitting group members to update the group state in a consistent, fault-tolerant manner. Moreover, although we've described the virtual synchrony model in pictures, it can also be expressed as a set of temporal logic equations. For our purposes in this chapter,[2] we won't need that sort of formalism, but readers can find temporal logic specifications of the model in [72, 25].

6.2 Distributed Consistency: Who Needs It?

Virtual synchrony guarantees a very powerful form of distributed, fault-tolerant consistency. With a model such as this, applications can replicate objects (individual variables, large collections of data items, or even files or databases), track their evolving state and cooperate to perform such actions as searching in parallel for items within the collection. The model can also easily support synchronization by locking, and can even provide distributed versions of counting semaphores, monitor-like behavior, etc. But not every replicated service requires the sorts of strong guarantees that will be our focus here, and virtual synchrony isn't the only way to provide them.

Microsoft's scalable cluster service uses a virtual synchrony service at its core [54], as does IBM's DCS system, which provides fault-tolerant management and communication technology for WebSphere and server farms [30, 29]. Yahoo's Zookeeper service [64] adopts a closely related approach. Google's datacenters are structured around the Google File System which, at its core, depends on a replicated "chunk master" service (it uses a simple primary/backup scheme), and a locking service called Chubby [19, 21]. The Chubby protocols were derived from Lamport's Paxos algorithm [48]. Most Google applications depend on Chubby in some way: some share a Chubby service, others instantiate their very own separate Chubby service and use it privately, while others depend on services like Big Table or MapReduce, and thus (indirectly) on Chubby. But not all roads lead to state machines. HP's

[1] The highest event rates are reached when events are very small and sent asynchronously (without waiting for recipients to reply). In such cases an implementation can pack many events into each message it sends. Peak performance also requires network support for UDP multicast, or an efficient overlay multicast.

[2] We should note that [72] is a broad collection of Isis-related papers and hence probably the best reference for readers interested in more detail. A more recent text [11] covers the material reviewed in this chapter in a more structured way, aimed at advanced undergraduates or graduate students.

Sinfonia service implements a distributed shared memory abstraction with transactional consistency guarantees [1].

The need for consistent replication also arises in settings outside of data centers that support cloud computing. *Edge Computing* may be the next really big thing: this involves peer-to-peer technologies that allow applications such as the widely popular Second Life game to run directly between client systems, with data generated on client machines or captured from sensors transmitted directly to applications that consume or render it [60]. Although highly decentralized, when edge computing systems need consistency guarantees, they require exactly the same sorts of mechanisms as in the datacenter services mentioned above. On the other hand, many peer-to-peer applications manage quite well without the forms of consistency of interest here: Napster, Gnutella, PPLive and BitTorrent all employ stochastic protocols.

6.3 Goals in This Chapter

Whether one's interest is focused on the cloud, looks beyond it to the edge, or is purely historical, it makes sense to ask some basic questions. What sorts of mechanisms, fundamentally, are needed, and when? How were these problems first identified and solved? What role does the classic consensus problem play? What are the arguments for and against specific protocol suites, such as virtual synchrony or Paxos? How do those protocol families relate to one-another?

This article won't attempt to answer all of those questions; to do so would require a much longer exposition than is feasible here, and would also overlap other articles in this collection. As the reader will already have gathered, we'll limit ourselves to virtual synchrony, and even within this scope, will restrict our treatment. We'll try to shed light on some of the questions just mentioned, and to record a little snapshot of the timeline in this part of the field. For reasons of brevity, we won't get overly detailed, and have opted for a narrative style rather light on theoretical formalism. Moreover, although there were some heated arguments along the way, we won't spend much time on them here. As the old saying goes, academic arguments are especially passionate because the underlying issues are so unimportant!

6.4 Historical Context

Virtual synchrony arose in a context shaped by prior research on distributed computing, some of which was especially influential to the model, or to the Isis Toolkit architecture:

1. Leslie Lamport's seminal papers had introduced theoretical tools for dealing with time in distributed systems — and in the process, suggested what came to be known as the "replicated state machine" approach to fault-tolerance, in which a deterministic event-driven application is replicated, and an atomic broadcast primitive used to drive its execution. Especially relevant were his 1978 paper, which was mostly about tracking causality with logical clocks but introduced

state machine replication in an example [45], and his 1984 paper, which explored the approach in greater detail [46]. Fred Schneider expanded on Lamport's results, showing that state machine replication could be generalized to solve other problems [68].

2. The Fischer, Lynch and Patterson result proved the impossibility of asynchronous fault-tolerant consensus [35]. One implication is that no real-world system can implement a state machine that would be guaranteed to make progress; another is that no real system can implement an accurate failure detector. Today, we know that most forms of "consistency" for replicated data involve solving either the consensus problem as originally posed in the FLP paper, or related problems for which the impossibility result also holds [20, 25].
3. On the more practical side of the fence, Cheriton, Deering and Zwaenepoel proposed network-level group communication primitives, arguing that whatever the end-to-end abstraction used by applications, some sort of least-common denominator would be needed in the Internet itself (this evolved into IP multicast, which in turn supports UDP multicast, much as IP supports UDP). Zwaenepoel's work was especially relevant; in [24] he introduced an operating-system construct called a "process group", and suggested that groups could support data replication, although without addressing the issue of replication models or fault-tolerance.
4. Database transactions and the associated theory of transactional serializability were hot topics. This community was the first to suggest that replication platforms might offer strong consistency models, and to struggle with fundamental limits. They had their own version of the FLP result: on the one hand, the fault-tolerant "available copies" replication algorithm, in which applications updated replicas using simple timeout mechanisms for fault-tolerance, was shown to result in non-serializable executions [5]. On the other, while quorum mechanisms were known to achieve 1-copy serializability [4], they required two-phase commit (2PC) protocols that could block if a failure occurred. Skeen proposed a three-phase commit (3PC) [70]: with a perfect failure detector, it was non-blocking. (The value of 3PC will become clear later, when we talk about group membership services.)
5. Systems such as Argus and, later, Clouds were proposed [49, 55]. The basic premise of this work was that the transactional model could bring a powerful form of fault-tolerance to the world of object-oriented programming languages and systems. A criticism of the approach was that it could be slow: the methodology brings a number of overheads, including locking and the need to run 2PC (or 3PC) at the end of each transaction.

All of this work influenced the virtual synchrony model, but the state machine model [45, 46, 68] was especially important. These papers argued that one should think of distributed systems in terms of event orderings and that doing so would help the developer arrive at useful abstractions for fault-tolerance and replication. The idea made sense to us, and we set out to show that it could have practical value in real systems.

To appreciate the sense of this last remark, it is important to realize that in 1983, state machine replication meant something different than it does today. Today, as many readers are probably aware, the term is used in almost any setting where a system delivers events in the same order to identical components, and they process them deterministically, remaining in consistent states. In 1983, however, the state machine model was really offered as an illustration of how a Byzantine atomic broadcast primitive could be used in real applications. It came with all sorts of assumptions: the applications using state machine replication were required to be deterministic (ruling out things like threads and exploitation of multicore parallelism), and the network was assumed to be synchronous (with bounded message delays, perfectly synchronized clocks, and a way to use timeout to sense failures). Thus, state machine replication was really a conceptual tool of theoretical, but not practical, value at the time the virtual synchrony work began. This didn't change until drafts of the first Paxos paper began to circulate in 1990 [48], and then Paxos was used as a component of the Frangiapani file server in 1997.

In our early work on virtual synchrony, we wanted to adapt the state machine concept of "ordered events" to practical settings. Partly, this involved reformulating the state machine ideas in a more object oriented manner, and under assumptions typical of real systems. But there was also the issue of the Byzantine atomic broadcast protocol: a very slow protocol, at least as the community understood such protocols at the time (faster versions are common today). Our thinking led us to ask what other sorts of fault-tolerant multicast protocols might be options.

This line of reasoning ultimately took us so far from the state machine model that we gave our model its own name. In particular, virtual synchrony weakened the determinism assumptions, targeted asynchronous networks, added process groups with completely dynamic membership, and addressed network partitioning faults. All were innovations at that time. By treating process groups as replicated objects, we separated the thing being replicated (the object) from the applications using it (which didn't need to even be identical: a process group could be shared among an application coded in C, a second one coded in Ada, and a few others coded in C++). Groups could be used to replicate a computation, but also to replicate data, or even for purposes such as synchronization.

Today, as readers will see from other chapters in this collection, the distinctions just listed have been eroded because the two models both evolved over time (and continue to do so). The contemporary state machine approach uses dynamic process group membership mechanisms very similar to those used in virtual synchrony. These mechanisms, however, were introduced around 1995, almost a decade after the first virtual synchrony papers were published. Virtual synchrony evolved too, for example by adding support for partitionable groups (work done by the Transis group; we'll say more about it later). Thus, today, it isn't easy to identify clear differences between the best replicated state machine implementations and the most sophisticated virtual synchrony ones: the approaches have evolved towards one-another over the decades. But in 1983, the virtual synchrony work was a real departure from anything else on the table.

6.4.1 Resilient Objects in Isis V1.0

We've summarized the background against which our group at Cornell first decided to develop a new system. Staying with the historical time-line, it makes sense to discuss this first system briefly: it had some good ideas that lived on, although it also embodied a number of questionable decisions. This first system was called Isis (but not the Isis "Toolkit"), and was designed to support something we called *resilient objects*. The goal was to help developers build really fast, fault-tolerant services.

Adopting what was then a prevailing paradigm, Isis V1.0 was a translator: it took simple object-oriented applications, expressed in a language similar to that of Argus or Clouds, and then translated them into programs that could run on multiple machines in a network, and would cooperate to implement the original object in a fault-tolerant, replicated manner. When an application issued a request to a resilient object, Isis would intercept the call, then distribute incoming queries in a way that simultaneously achieved high availability and scalable performance [16, 8]. The name Isis was suggested by Amr El Abbadi, and refers to an Egyptian resurrection myth in which Isis revived Osiris after he had been torn apart by his enemy, Seth. Our version of Isis revived resilient objects damaged by failure.

In retrospect, the initial version of Isis reflected a number of misconceptions on our part. Fortunately, it wasn't a complete wash: in building the system, we got one thing right, and it had a huge impact on the virtual synchrony model. Isis dealt with failure detection in an unusual way, for the time. In most network applications, failures are detected by timeout at the network layer, and throw exceptions that are handled "end to end" by higher layer logic. No failure detector can achieve perfect accuracy, hence situations can arise in which processes p, q, and r are communicating, and p believes that q has failed — but r might still believe both are healthy. Interestingly, this is almost exactly the scenario that lies at the core of the problem with the transactional available copies replication scheme. Moreover, one can provoke such a problem easily. Just disrupt your local area network. Depending on the value of the TCP_KEEPALIVE parameter, connections will begin to break, but if the network outage is reasonably short, some connections will survive the outage, purely as a random function of when the two endpoints happen to have last exchanged messages or acknowledgements. This illustrates a pervasive issue: timeouts introduce inconsistency. FLP teaches us that the problem is fundamental.

Transactional systems generally overcome such problems using quorum methods, but Isis adopted a different approach: it included a separate *failure detection service*. When an Isis component detected a timeout, rather than severing the associated connection, it would complain to the failure detection service (which was itself replicated using a fault-tolerant protocol [15]). This group membership service (GMS) virtualized the notion of failure, transforming potentially inaccurate failure suspicions into what the system as a whole treated as bedrock truth. Returning to our example above, p would report q as faulty, and the service would dutifully echo this back out to every process with a connection to q. The word of this detection service was authoritative: once it declared a component faulty, the remainder of our system believed the declaration and severed connections to q. If a mistake occurred

and process q was still alive, q would be forced to rejoin the system much like a freshly-launched process. In particular, this entails rejoining the process groups to which it previously belonged, and reinitializing them.

Today, it would be common to say that Isis implemented a *fail-stop* model [69]: one in which processes fail by halting, and where those failures are detectable. In effect, the Isis GMS creates a virtual network abstraction, translating imprecise timeouts into authoritative failure events, and then notifying all components of the system so that they can react in a coordinated way. This simplifies the design of fault-tolerant protocols, although they remain challenging to prove correct.

The reader may be puzzled by one issue raised by this approach. Recall from the introduction that we need to avoid split-brain behavior, in which a system becomes logically partitioned into two or more subsystems that each think the other has failed, and each think themselves to be running the show. We mentioned that the GMS itself was replicated for high availability. How can the GMS itself avoid split-brain failures?

Isis addressed this by requiring a form of rolling majority consent within the GMS. Membership in the service was defined as a series of membership epochs — later, we began to use the term "view."[3] To move from view i to view $i+1$, a majority of the GMS processes in view i were required to explicitly acknowledge view $i+1$. The protocol was initiated by the oldest GMS process still operational, and requires a 2PC as long as the leader is healthy. If any process suspects the current leader of being faulty, it can trigger a 3PC whereby the next oldest process replaces the apparently faulty one as leader. Our 1985 SOSP paper focused on the system issues and performance [8]; a technical report gave the detailed protocols [13], and later those appeared as [15]. In a departure from both the Byzantine Agreement work and the Consensus model used in FLP, Isis made no effort to respect any sort of ground-truth about failures. Instead, it simply tried to detect real crash failures quickly, without making too many mistakes.[4]

In adopting this model, Isis broke new ground. Obviously, many systems developed in that period had some form of failure detection module. However, Isis used

[3] Isis was the first to use this term, which was intended as an allusion to "dynamically materialized views", a virtualization mechanism common in relational database systems: the user poses a standing query, and as the database is updated, the result of the query is continuously recomputed. Queries treat the resulting relation as if it were a real one. At the time, we were thinking of the membership of a group as a sequence of records: membership updates extend the sequence, and multicast operations read the current membership and deliver messages to the operational processes within it. In effect, a multicast is delivered to a "dynamically materialized view of the membership sequence" containing the target processes. The term was ultimately adopted by many other systems.

[4] Obviously, this approach isn't tolerant of malicious behavior: any mistaken failure detection could force an Isis process to drop out of the system and then rejoin. Our reasoning was pragmatic: Isis was a complex system and early versions were prone to deadlock and thrashing. We included mechanisms whereby a process would self-check and terminate itself if evidence of problems arose, but these didn't always suffice. By allowing any process to eject any other process suspected as faulty, Isis was able to recover from many such problems. The obvious worry would be that a faulty process might start to suspect everyone else, but in practice, this sort of thing was never observed.

its membership module throughout, and the membership protocol can be recognized as a fault-tolerant agreement (consensus) solution.

Today, this mechanism may seem much less novel. For example, contemporary implementations of the state machine approach, such as the modern Paxos protocols, have a dynamically tracked notion of membership (also called a view), and use a leader to change membership. However, as noted earlier, when Paxos was introduced in 1990, the protocol wasn't leader-based: it assumed a fixed set of members, and all of them had perfectly symmetric roles. Leaders were introduced into Paxos much later, with a number of performance-enhancing optimizations. Thus when Isis introduced the approach in 1983, it was the first system to use this kind of dynamic membership tracking.

In adopting this approach, we rejected a tenet of the standard Internet TCP protocol: in keeping with the end-to-end philosophy, TCP (and later, early RPC protocols) used timeouts to detect failures in an uncoordinated manner. We also departed from the style of quorum-based update used in database systems, where the underlying set of nodes is fixed in advance (typically as a set of possible participants, some of which might be unavailable from time to time), and where each update must run as a 2PC: a first phase in which an attempt is made to reach a write-quorum of participants, and a second phase in which the participants are told if the first phase succeeded. As we'll see momentarily, the cheapest virtually synchronous multicast avoids this 2PC pattern and yet still ensures that delivery will occur within the primary partition of the system: not the identical guarantee, but nonetheless, very useful.

With the benefit of hindsight, one can look back and see that the convergence of the field around uncoordinated end-system based failure detection enshrined a form of inconsistency into the core layers of almost all systems of that period. This, in turn, drove developers towards quorum-based protocols, which don't depend on accurate failure detection — they obtain fault-tolerance guarantees by reading and writing to quorums of processes, which are large enough to overlap. Yet as we just saw, such protocols also require a two phase structure, because participants contacted in the first phase don't know yet whether a write quorum will actually be achieved. Thus, one can trace a line of thought that started with the end-to-end philosophy, became standardized in TCP and RPC protocols, and ultimately compelled most systems to adopt quorum-based replication. Unfortunately, quorum-based replication is very slow when compared with unreliable UDP multicast, and this gave fault-tolerance a bad reputation. The Isis protocols, as we've already mentioned, turned out to do well in that same comparison.

We've commented that a GMS greatly simplifies protocol design, but how? The key insight is that in a failstop setting, protocols benefit from a virtualized environment where processes appear to fail by halting, and where failures are reported as an event, much like the delivery of a "final message" from the failed process (in fact, Isis ignored messages from processes reported as faulty, to ensure that if a failure was transient, confusion couldn't arise). For example, it became safe to use the available copies replication scheme, an approach that risks non-serializable executions when timeouts are used to detect failures. Internally, we were able to use

protocols similar in style to Skeen's 3PC, which is non-blocking with "accurate" failure detections.

Above, we indicated that this article won't say very much about the various academic arguments that erupted around our work. It is interesting, however, to realize that while Isis drew on ideas from many research communities, it also had elements that were troubling to just about every research community of that period. We used a technology reminiscent of state machines, but in a non-Byzantine setting. We use terminology close to that of the consensus literature, but proposed a solution in which a healthy process might be treated as faulty and forced to restart, something that a normal consensus definition wouldn't allow. Our GMS service violated the end-to-end approach (network-level services that standardize failure detection are the antithesis of end-to-end design). Finally, we claimed that our design was intended to maximize performance, and yet we formalized the model and offered protocols with (partial) correctness proofs. Not surprisingly, all of this resulted in a mixed reception.

6.4.2 Beyond Resilient Objects

As it turned out, resilient objects in Isis V1.0 weren't much of a success even relative to our own goals. Beyond its departures from the orthodoxies of the period, the system itself had all sorts of problems. First, resilient objects used a transactional programming language similar to the ones used by Argus and Clouds. However, whereas those systems can now be understood as forerunners of today's transactional distributed computing environments and software transactional memories, Isis was aimed at what we would now call the cloud computing community. To convince users that this language was useful, we needed to apply it to network services such as load balancers, DNS resolvers, etc. But most such services are implemented in C or C++, hence our home-brew language seemed unnatural. Moreover, it turned out to be difficult to adapt the transactional model for such uses.

The hardest problems relate to transactional isolation (the "I" in the ACID model). In a nutshell, transactional systems demand that uncommitted actions be prevented from interacting. For example, if an uncommitted transaction does a DNS update, that DNS record must be viewed as provisional. Until the transaction commits or aborts, other applications either can't be allowed to look at it or, if they "optimistically" read the record, the readers become dependent upon the writer.

This may seem straightforward, but creates a conundrum. Locking records in a heavily-used service such as the DNS isn't practical. But if such records aren't locked, long dependency chains arise. Should an abort occur, it may cascade through the system. Moreover, no matter how one implements concurrency control, it is hard to achieve high performance unless transactions are very short-lived. This forces applications to use lots of very short atomic actions, and to employ top-level actions whenever possible. But such steps "break" the transactional model. There was a great deal of work on this issue at the time (the Argus team had one approach, but it was just one among many: others included Recovery Blocks [63] and Sagas

[26]). None of these solutions, however, appeared to be well matched with our target environment.

Faced with these issues, it occurred to us that perhaps the core Isis infrastructure might be more effective if we unbundled it and offered it as a non-transactional library that could be called directly from C or C++. Of course, the system had been built to support transactions, and our papers had stressed transactional consistency models. This led us to think about what it would mean to offer a "transactional process group" in which we could retain strong consistency and fault-tolerance properties, but free applications from the problematic consequences of the transactional model.

The key idea was to think of the membership of each group as a kind of shared database that would be updated when processes joined and left the group, and "read" by multicast protocols, resulting in a form of transactional serializability at the level of the multicasts used to send updates to replicated data. This perspective led us to the virtual synchrony model. Stripped down versions of the model were later proposed, notably "view-atomic multicast" as used by Schiper and Sandoz [66] and "view synchrony", proposed by Guerraoui and Schiper in [36] (the Isis literature used the term "virtually synchronous addressing" for this property). In [3], Babaoglu argued that view synchrony should be treated as the more fundamental model, and developments have tended to reinforce this perspective.

6.4.3 The Isis Toolkit and the Virtual Synchrony Model

Accordingly, we set out to re-implement the Isis system as a bare-bones infrastructure that would present a "toolkit" API focused on processes that form groups to replicate data, back one-another up for fault-tolerance, coordinate and synchronize their actions, and perform parallel operations such as concurrent search of large databases.[5] Other tools within the toolkit offered access to the group membership data structure, delivered event upcalls when membership changed, and supported state transfer. A "news" service developed by Frank Schmuck provided topic-oriented publish/subscribe. None of these services was itself transactional, but all gained consistency and fault-tolerance from the underlying model. Isis even included a conventional transactional subsystem (nobody used it).

Of course, our goal wasn't just to make our own tools fault-tolerant: we wanted to make the life the application developer simpler, and for this reason, the virtual synchrony model was as much a "tool" as the ones just listed: those were library tools, while the model was more of a conceptual tool. As we saw in the introduction, a virtually synchronous system is one indistinguishable from a synchronous one. This is true of applications built using virtual synchrony too: the developer starts with a very synchronous design, and is assisted in developing a highly con-

[5] The system also included a home-brew threads package, and a standard library for serializing data into messages and extracting data from them. Cthreads weren't yet available, and we learned later that quite a few of the early Isis users were actually looking for a threads package when they downloaded the toolkit!

current, performance-efficient solution that retains the simplicity and correctness characteristics of the original synchronous version.

The key to this methodology is to find systematic ways that event ordering can be relaxed, leaving the platform the freedom to deliver some messages in different orders at different group members. We'll discuss the conditions under which this can happen below, but the essential idea is to allow weaker delivery orderings when the delivery events commute, so that the states of the members turn out to be identical despite the different event orderings. One benefit of this approach is to reduce the risk of a "poison pill" scenario, in which a state-sensitive bug might cause all members of a traditional state-machine replicated service to crash simultaneously. In virtual synchrony, the members of a group are in equivalent states, but recall that a group is usually a replicated object: Isis rarely replicated entire processes. Thus the processes joining a group might actually differ widely: they could be coded in different languages, may have joined different sets of additional groups, and their executions could be quite non-deterministic. In contrast the code implementing a typical object replica might be very small: often, just a few lines of simple logic. All of this makes it much less likely that a single event will cause many members to crash simultaneously.

Another difference is visible at the "end" of the execution, on the right: two processes become partitioned from the others, and continue to exchange some messages for a short while before finally detecting their isolated condition and halting. Although the application developer can ask Isis not to allow such runs (they use the *gbcast* primitive to send messages, or invoke the *flush* primitive before delivering messages), the default allows them to arise for bounded periods of time.[6] These messages may never be delivered at all in the other processes, and if they are, may not be delivered in the order seen by the processes that failed. Given an application that can tolerate these kinds of minor inconsistencies, Isis gained substantial performance improvements by permitting them. Moreover, by working with application developers, we discovered that stronger guarantees are rarely required. Often, an application that seems to need strong guarantees can easily be modified into one for which weaker guarantees suffice.[7]

One proves that a system implements virtual synchrony by looking at the runs it can generate. Given a run, the first step is to erase any invisible events — events that occurred at processes that later failed, and that didn't have a subsequent causal

[6] The internal timeout mechanisms mentioned earlier ensure that an isolated process would quickly discover the problem and terminate itself; developers could fine-tune this delay.

[7] Few Isis applications maintained on-disk state that couldn't be discarded after a restart. For example, consider a load-balancing service, in which nodes report their loads through periodic multicasts, and assign new tasks to lightly-loaded nodes. Now suppose that a node running the service crashes and later restarts. It won't need any state from prior to the crash: a state-transfer can be used to bring it up to date. The example illustrates a kind of multicast that reports a transient state change: one that becomes stale and is eventually forgotten as the system evolves over time. Experience with Isis suggested that these kinds of multicasts are not merely common, but constitute the overwhelming majority of messages transmitted within applications. The insight here is that even if a transient multicast is delivered non-atomically, the service using the multicast might not be at risk of user-visible inconsistencies.

path to the processes that survived. Next, where events are known to commute, we sort them. If we can untangle Figure 6.2 and end up with Figure 6.1, our run was "indistinguishable" from a synchronous run; if *all* runs that a protocols permits are indistinguishable from synchronous runs, it is virtually synchronous.

As discussed earlier, network partitioning is avoided by requiring that there can only be one "primary" partition active in the network. In Figure 6.2, the majority of the processes are on the side of the system that remains active. The isolated processes are too few in number to form a primary partition, and will quickly discover this problem and then shut down (or, in fancier applications, shift to a "disconnected" mode of operation).[8] A special mechanism was used to handle "total" failure, in which the primary partition is lost. Basically, the last processes to fail are able to restart the group, resuming execution using whichever state reflects the most updates. Although the problem is difficult in general settings [71], in a virtual synchrony environment identifying these last failed processes becomes easy if we simply log each group view.

6.4.4 A Design Feature Motivated by Performance Considerations

The most controversial aspect of virtual synchrony centers on the willingness of the system to deliver unstable events to applications, despite the risk that a failure might "erase" all evidence that this occurred. Doing so violates one of the tenants of the Consensus model as articulated by the FLP paper: the *uniform agreement property*, which requires that if one process *decides* $v \in \{0, 1\}$, then every non-faulty process that decides, decides *v*. As stated, this implies that even if a process decides and then crashes immediately, the rest of the system will make the identical decision value. Paxos, for example, provides this guarantee for message delivery, as does the uniform reliable multicast [65]. Moreover, virtual synchrony sometimes does so as well: this is the case for process group views, uniform multicast protocols, and for events delivered using *gbcast*. Why then did we offer a "broken" multicast primitive as our default mode of operation?

To understand our reasoning, the reader will need to appreciate the emphasis on performance that dominated the systems community during that period, and continues to dominate today. For the networking community, there will never be a point at which the network is "too fast" to be seen as a bottleneck. Even our earliest papers came under criticism because reviewers argued that in the real world, no protocol slower than UDP multicast would be tolerated. Yet UDP multicast is a hardware-supported unreliable protocol in which the sender sends a message, and "one hop downstream", the receivers deliver it! Competing with such a short critical path creates all sorts of pressures. The features of the virtual synchrony model,

[8] The developer controlled the maximum delay before such a problem would be detected. By manipulating timeout parameters, the limit could be pushed to as little as three to five seconds. Modern machines are faster, and today the limit would be a small fraction of a second. By using *gbcast* or *flush*, "lost events" such as the ones shown in Figure 6.2 are eliminated, but performance is sharply reduced.

taken as a whole, represent a story that turned out to be competitive with this sort of raw communication primitive: most virtually synchronous multicasts could be sent asynchronously, and delivered immediately upon receipt, just as would be the case if one were using raw UDP multicast. This allowed us to argue that users could have the full performance of the hardware, and yet would also gain much stronger fault-tolerance and consistency semantics.

As we've emphasized, an application can be designed so that if a multicast needs the stronger form of safety, in which any multicast is delivered to all operational processes, or to none, the developer simply sends it with a uniform multicast primitive, or with *gbcast*, or invokes *flush* prior to delivery. But our experience with Isis revealed that this is a surprisingly rare need. The common case was simply to send the multicast unsafely. Doing so works because the great majority of multicasts either don't change the application state at all, or update what can be understood as "transient" state, relevant to the system for a short period of time, but where persistency isn't needed. In such cases, it may not matter if a failed process received a multicast that nobody else will receive, or so an unusual event ordering: if it ever recovers, nobody will notice that immediately before crashing, it experienced a strange sequence of events.

For example, a query might be multicast to a group in order to request some form of parallel search by its members. Queries don't change application state at all, so this kind of multicast can certainly be delivered without worrying about obscure failure cases.

Many kinds of updates can be sent with a non-uniform multicast primitive, too. Probably the best example is an update to a cache. If a process restarts from a crash, it certainly won't assume that cached data is currently accurate; either it will validate cached items or it will clear the cache. Thus a really fast, reliable, ordered multicast is exactly what one wants for cache updates; uniform delivery simply isn't needed. Other examples include updates that only touch transient state such as load-balancing data, and internal chit-chat about the contents of transient data structures such as a cache, a lock queue or a pending-task queue (a "work queue"). On recovery from failure, a process using such a data structure will reinitialize itself using a state transfer from an operational process. If the whole group fails, we either restart in a default state, or have one of the last processes to fail restart from a checkpoint.

The application that really needs uniform delivery guarantees, because it maintains persistent state, would be a big on-disk database. Obviously, databases are important, but there aren't many more such examples. Our point, then, is that this category is relatively small and the stronger guarantees they need are costly. In Isis, we simply made the faster, more common multicast the default, and left it to the developer to request a stronger guarantee if he or she needed it. In contrast, the Paxos protocols offer the stronger but more costly protocol by default, whether needed or not.

6.5 Dynamic Membership

Let's revisit the features of the model somewhat more carefully. As we've seen, the basic idea of replicating data within a group of processes traces to Lamport's state machine concept. In addition to removing state machines from the Byzantine world where they were first proposed, Isis departed from Lamport's work in several ways.

We've already seen one departure, namely our use of a dynamic membership model. At the time we built Isis, one might think that a static model of the sort used in databases (and later in Paxos) would have seemed more natural, but in fact our model of how process groups would be used made dynamic membership seem much more obvious. After all: we assumed that applications might use large numbers of groups, because for us, the granularity of a group was a single object, not an entire application. Like files that applications open, access, then close, we saw groups as shared structures that applications would join, participate in for a while, and then depart from. With completely dynamic membership, a group becomes an organic abstraction that can, in effect, wander around the system, residing at processes that are currently using it, but over time, moving arbitrarily far from the initial membership.

Of course, we realized that some systems have groups of server platforms and need to know that groups will always contain a majority of the servers (databases often require this). In fact, Isis supported both models. Implicit in the normal behavior was a *weighting function* and a *minimum group commit weight*. By default, the weighting function weighted all processes 1.0, and used a minimum commit weight of 0.0, but it was possible to override these values, in which case no new view could be committed unless a majority of the members of the previous view had consented to it *and* the sum of weights of the new group view exceeded the minimum. Thus, to ensure that the majority of some set of k servers would always be present in each new group view, one simply told Isis to weight the servers 1.0 and all non-servers 0.0, and then specified that new views have a weight greater than $k/2$.

6.5.1 Local Reads and Fast Updates

Dynamic membership is the key to an important performance opportunity: many of the protocols we were competing with at the time assumed that their role was to replicate some service at a statically defined set of replicas, and used quorum methods to do both reads and updates. To tolerate failures, even reads needed to access at least two members, since any single member might have been down when an update was done and hence have a stale state. By tracking membership dynamically, in a setting where a trusted primary-partition GMS reports liveness, we could be sure that every member of a group was also up to date, and reads could then be done entirely locally. In [38] we showed this, and also gave a locking protocol in which read-locks are performed locally. Thus, reads never require sending messages, although updates obviously do — for locking, to communicate the changes to data, and for the commit protocol when the transaction completes. The resulting protocol far outperforms quorum-based algorithms in any setting where reads are common,

or where updates are bursty. In the worst case, when updates are common and each transaction performs just one, performance is the same as for a quorum scheme.

The key insight here is that within a virtual synchrony system, the group view represents a virtual world that can be "trusted". In the event of a partitioning of the group, processes cut off from the majority might succeed in initiating updates (for example if they were holding a lock at the time the network failed), but would be unable to commit them — the 2-phase protocol would need to access group members that aren't accessible, triggering a view change protocol that would fail to gain majority consent. Thus any successful read will reflect all prior updates: committed ones by transactions serialized prior to the one doing the read, plus pending updates by the reader's own transaction. From this we can prove that our protocol achieves one-copy serializability when running in the virtual synchrony model. And, as noted, it will be dramatically faster than a quorum algorithm achieving the identical property.

This may seem like an unfair comparison: databases use quorums to achieve serializability. But in fact Isis groups, combined with locking, also achieve serializability. Because the group membership has become a part of the model, virtually synchronous locking and data access protocols guarantee that any update would be applied to all replicas and that any read-locked replica reflects all prior updates. In contrast, because quorum-based database systems lack an agreed-upon notion of membership, to get the same guarantees in the presence of faults, a read must access two or more copies: a read quorum. Doing so is the only way to be sure that any read will witness all prior updates.

Enabling applications to read a single local replica as opposed to needing to read data from two or more replicas, may seem like a minor thing. But an application that can trust the data on any of its group members can potentially run any sort of arbitrary read-oriented computation at any of its members. A group of three members can parallelize the search of a database with each member doing 1/3 of the work, or distribute the computation of a costly formula, and the code looks quite normal: the developer builds any data structures that he or she likes, and accesses them in a conventional, non-distributed manner. In contrast, application programmers have long complained about the costs and complexity of coding such algorithms with quorum reads. Each time the application touches a data structure, it needs to pause and do a network operation, fetching the same data locally and from other nodes and then combining the values to extract the current version. Even *representing* data becomes tricky, since no group member can trust its own replicas. Moreover, whereas virtually synchronous code can execute in straight-line fashion without pausing, a quorum-read algorithm will be subjected to repeated pauses while waiting for data from remote copies.

Updates become faster, too. In systems where an update initiator doesn't know which replicas to "talk to" at a given point in time, there isn't much choice but to use some kind of scatter-shot approach, sending the update to lots of replicas but waiting until a quorum acknowledged the update before it can be safely applied. Necessarily, such an update will involve a 2PC (to address the case in which a quorum just can't

be reached). In virtual synchrony, an update initiated by a group member can be sent to the "current members", and this is a well-defined notion.

6.5.2 Partitionable Views

This discussion raises questions about the conditions under which progress can be guaranteed during partitioning failures. Danny Dolev's research group became fascinated with the topic and went much further with it than we ever did at Cornell. Dahlia Malkhi, visiting with us in 1992, helped formalize the Isis model; the model in Chapter 6 of the book we published on the Isis system was due to her [72]. Upon returning to Hebrew University, she was the lead developer for the Transis [31] system, sharing some code with our Horus system, but using her own GMS protocols redesigned to maximize availability during partitioning failures, and including multicast protocols that can be traced back to the UCSB Totem project. The Transis protocol achieves the highest possible availability during partitioning failures [42]. However, this author always found the resulting model tricky to work with, and it was not widely adopted by application developers. Subsequent work slightly simplified the model, which became known as *extended view synchrony* [57], but it remains hard to develop non-trivial applications that maximize availability during partitioning failures.

6.6 Causally Ordered Multicast: *cbcast*

Dynamic membership only addresses some costs associated with multicasts that carry updates. In the timeframe when we developed our update protocols, the topic was very much in the air. Lamport's papers had been followed by a succession of theoretical papers proposing all sorts of protocols solving such problems as Byzantine Agreement, totally ordered atomic broadcast, and so forth — again, within static groups. For example, one widely cited paper was the Chang and Maxemchuck protocol [22], which implemented a totally ordered atomic multicast that used a circulating token to order messages. To deliver an update, one might have to wait for the token to do a full circuit of a virtual ring linking the group members. Latency increased linearly in the size of the group: a significant cost for large groups, but tolerable for a group with just two or three members.

Our initial work with Isis used a totally ordered protocol proposed by Dale Skeen and based on a similar, earlier, protocol by Leslie Lamport: it involved a 2PC in which logical timestamps were exploited to order multicasts [15, 45]. This was faster than most other total ordering protocols, but still was potentially as slow as the slowest group member. We wanted to avoid the two-phase flavor that pervade such protocols, and became interested in protocols that exploited what might be called application-specific optimizations. For example, knowing that the sender of a multicast holds a mutually exclusive lock within a group, a totally ordered multicast can be built using a protocol with the "cost" of a sender-ordered (FIFO) multicast. Frans Kaashoek, who ultimately wrote his PhD thesis on this topic [41], showed

that token-based protocols of this sort have all sorts of other advantages too, including better opportunities to aggregate small messages into big ones, the possibility of asynchronous garbage collection, and also match well with applications that produce updates in a bursty pattern.

In our work, we realized that FIFO order has a generalization that can extend the power of the multicast primitive at surprisingly low cost — just a few bytes per message. The trick is to put a little extra ordering information onto the message (the solution we ultimately favored used a small *vector timestamp* with a single counter per member of the current group view [17]). The rule for delivering a message generalizes the rule used for FIFO ordering: "if message x arrives and its header indicates that there is a prior message y, delay x until y has been delivered". But now, "prior" is interpreted using the vector timestamp ordering rule, rather than the usual comparison of sender sequence numbers.

Isis used this approach to support a protocol we called *cbcast*: a reliable, view-synchronous multicast that respected the potential causality ordering (the transitive closure of the FIFO ordering). One way to visualize this ordering property is to think about a system in which process p does some work, and then sends an RPC to process q asking it to do a subtask, and so forth. When the RPC returns, p resumes working. Now suppose that "work" involves sending multicasts. A FIFO ordering would deliver messages from x in the order it sent them, and similar for y, but a node receiving messages from both p and q could see them in an arbitrary order. We see this in the figure below; the heavy lines denote the "thread of execution".

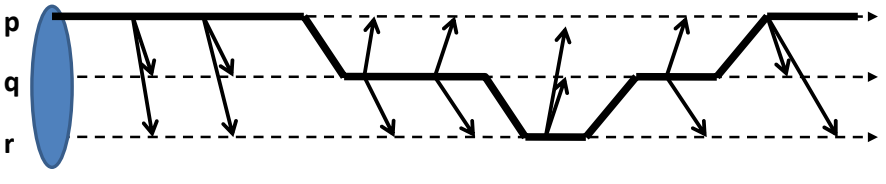

Fig. 6.3 Causally ordered multicasts.

One way to think about Figure 6.3 is to imagine that process p "asked" q to do that work, and q in turn issued a request to r. In some sense, q is a continuation of a thread of execution running in p. The figure highlights this visually: the single thread of control is the one shown in dark black, first running at p, then q, then r, and then finally returns back to p. The *cbcast* primitive respects ordering along this kind of thread of control. If multicast b (perhaps sent by q) could be causally ordered after multicast a (perhaps sent by p), a FIFO ordering won't necessarily deliver the messages in the order they were sent because they had different senders. In contrast, *cbcast* will deliver a before b at any destinations they share. The idea is very intuitive if one visualizes it this way.

Performance for *cbcast* can be astonishingly good: running over UDP multicast, this primitive is almost as fast as unreliable UDP multicast [17]. By using *cbcast*

to carry updates (and even for locking, as discussed in [39]), we accomplished our goal, which was to show that one could guarantee strong reliability semantics in a system that achieved performance fully comparable to that of Zwaenepoel's process groups running in the V system.

6.7 Time-Critical Applications

With its focus on speed, one major class of Isis applications turned out to involve systems that would traditionally have been viewed as "real-time" by the research community. As a result, Isis was often contrasted with the best known protocols in the fault-tolerant real-time community, notably the so-called Δ-T fault-tolerant broadcast protocols developed by Flaviu Cristian and his colleagues [27].

These protocols work in the following manner. The designer starts by specifying bounds on the numbers and types of failures that can occur (process failures, packet loss, clock failures). They also bound delays for packet delivery and clock skew by correct processes. Then, through multiple all-to-all broadcast stages, each multicast is echoed by its receivers until one can prove that at least one round included a correct sender and experienced no network faults (in effect: there must be enough rounds to use up the quota of possible failures). Finally, received multicasts are delayed for long enough to ensure that even if correct processes have worst-case clock skew and drift, they will still deliver messages in the same order and at roughly the same time as all other correct processes.

All of this takes time: at one workshop in the early 1990's, a speaker concerned about costs worked out the numbers for this and other broadcast protocols and argued that with as few as 10 processes under assumptions reasonable for that time, a Δ-T broadcast could take between 5 and 20 seconds to be delivered, depending upon the failure model selected (the protocols cover a range from fail-stop to Byzantine behavior). Most of the delay is associated with overcoming clock-drift and skew so as to deliver messages within a tight temporal window: the multicast relaying phases would normally run very quickly.

The strength of the Δ-T suite was its guarantee that messages will be delivered fault-tolerantly, in total order, *and within a bounded temporal delay* despite failures. On the other hand, these protocols lack the consistency property of virtual synchrony. For example, a "faulty" group member using the Δ-T protocols could miss a message, or deliver one out of order. This may not seem like a big deal until one realizes that a process can be temporarily faulty by simply running a bit slower than the bounds built into the system, or temporarily having a higher-than-tolerable clock skew. Since the Δ-T protocols have no explicit notion of group view, the protocols work around faults, rather than excluding faulty members. A discussion of the issue, with diagrams showing precisely how it can arise, can be found in [11].

Since no process can be sure it hasn't ever been faulty, no group member can ever be sure that its own data is current, because the protocol isn't actually required to operate correctly at faulty participants. This is a bit like obeying the speed limit without a reliable speedometer. One does the best one can, but short of driving absurdly slowly, there is a definite risk of briefly breaking the law. And indeed, driving

slowly is the remedy Δ-T protocol designers recommended: with these protocols, it was critical to set the parameters to very conservative values. One really doesn't want a correct process to be transiently classified as faulty; if that happens, all guarantees are lost.

Thus, builders of real-time systems who needed *provable* temporal guarantees, but could sacrifice speed and consistency, would find what they wanted in the Δ -T protocols. Isis, in contrast, offered much better performance and strong consistency, but without hard temporal delivery guarantees. The real-time community found itself immersed in a philosophical debate that continues to this day: Is real-time about predictable speed, or provable worst-case deadlines? The question remains unanswered, but Isis was used successfully in many sensitive settings, including air traffic control and process control in chemical refineries.

6.8 A Series of Commercial Successes, but Ultimately, a Market Failure

The combination of the virtual synchrony consistency model with an easily used toolkit turned out to be quite popular. Isis soon had large numbers of real users, who downloaded the free release from a Cornell web site. Eventually, the user base became so demanding that it made sense to launch a company that would do support, integration work and enhance the platform. Thus, the same protocols we designed and implemented at Cornell found their way into all sorts of real systems (details on a few can be found in [15] and [72]). These included the New York and Swiss Stock Exchange, the French Air Traffic Control System, the US Navy AEGIS, dozens of telecommunications provisioning systems, the control system of some of the world's largest electric and gas grid managers, and all sorts of financial applications. Many of these live on: today, the French ATC solution has expanded into many other parts of Europe and, to this author's knowledge, has never experienced a single problem. The New York Stock Exchange system operated without problems for more than a decade (they phased the Isis solution out in early 2007), running the fault-tolerant system that delivers data to the overhead displays and to external "feeds" like Reuters, Bloomberg and the SEC. During that decade, there were plenty of component crashes, but not a single disruption of actual trading.

Virtual synchrony was also adopted by a number of other research groups, including the Totem project developed by Moser and Melliar Smith [58], Dolev's Transis project [31], the European Phoenix project [52], Babaoglu's early e-Grid project, Amir's Spread system [2] (which continues to be widely used), and others. The UCSB team led a successful effort to create a CORBA fault-tolerance standard based on virtual synchrony. It offers lock-step state-machine replication of deterministic CORBA objects, and there were a number of products in the area, including Eternal [59], and Orbix+Isis, offered by IONA.

Unfortunately, despite these technical successes, virtual synchrony never became a huge market success [12]. The main commercial applications tended to be for replication of services, and in the pre-cloud computing data, revenue was mostly

generated on the "client side". The model continues to play an important role in many settings, but at the time of this writing there are only three commercial products using the model: JGroups, Spread and C-Ensemble. Of these, JGroups and Spread are the most widely used.

At Cornell, after completing Isis, we developed two Isis successors: first Horus [73], in which Van Renesse showed that a virtual synchrony protocol stack could be constructed as a composition of microprotocols (and set performance records along the way), and then Ensemble [37], a rewrite of Horus into O'CaML (a dialect of ML) by Mark Hayden. Ensemble was the basis of an interesting dialog with the formal type theory community. In a collaboration that drew upon an I/O Automata specification developed jointly by the Cornell team and Lynch's group at MIT, and used the Cornell NuPRL automated theorem proving system developed by Constable's group[50], a specification of many of the Ensemble protocols was created. NuPRL was then used to prove protocol properties and (through a form of partial evaluation) to generate optimized versions of the Ensemble protocols. Although the protocol stack as a whole was never "proved correct", the resulting formal structure was still one of the largest ever treated this way: the Ensemble protocol stacks that implement virtual synchrony included nearly 25,000 lines of code!

Virtual synchrony continues to play some important roles hidden within products that don't expose any form of directly usable group communication API. IBM has described patterning its DCS product on Ensemble [30, 29]. As mentioned early in this article, DCS is used for fault-tolerance and management layer in Websphere and in other kinds of services deployed within datacenters. We also worked with Microsoft to develop a scalable cluster management solution that ultimately shipped with the recent Longhorn enterprise server product; it runs large clusters and provides core locking and state replication services [54]. Again, a virtual synchrony protocol is used where strong consistency matters. Moreover, Google's Chubby and Yahoo's Zookeeper services both have structures strongly reminiscent of virtually synchronous process groups.

6.8.1 How Replication Was Used

In light of the focus on this volume on replication, it makes sense to review some of the uses to which these applications put the technology. Details can be found in [15] and [72].

- One popular option was to simply replicate some sort of abstract data type, in effect associating the object with the process group. In Isis, we saw two "styles" of data replication. For non performance-intensive uses, applications simply used the totally ordered multicast protocol, *abcast*, to propagate updates, and performed reads against any local copy. For performance-critical purposes, developers typically started with *abcast* but then optimized their applications by introducing some form of locking and then replacing the *abcast* calls with asyn-

chronous[9] *fbcast* (the FIFO multicast) or its cousin *cbcast*. One common pattern used *cbcast* for both purposes: to request and grant locks, and to replicate updates [39]. With the Isis implementation of *cbcast* this implements the Herlihy-Wing *linearizability* model [38].

- Replication was also applied to entire services. We expected that state-machine replication would be common among Isis users, but in fact saw relatively little use of this model until the CORBA "Orbix+Isis" product came out. Users objected to the requirement that applications be deterministic. The problem is that on modern platforms, concurrency through multithreading, timer interrupts, and non-deterministic I/O is so common that most developers couldn't develop a deterministic application even if they wanted to do so.
- Some applications used request replication for parallelism. Most servers are I/O bound, hence response time for many applications is limited by the speed with which a file or database can be searched. Many virtual synchrony applications replicate requests by multicasting them to a process group consisting of identical servers, which subdivided the work. For example, perhaps one server could search the first half of a database, and another the second half. This was a popular model, and is a very good match with search in datacenters, which often work with enormous files and databases. One can think of it as a very simple form of "map-reduce".
- Variations on primary-backup fault-tolerance were common. Isis users were loath to dedicate one machine to simply backing up another machine. However, the system also supported a generalization of primary-backup that we called "coordinator-cohort" that could be combined with a transparent TCP fail-over mechanism. In this model, each request was assigned to a different process group member, with another group member functioning as a backup, stepping in only if the primary crashed. The coordinator role was spread evenly within the group. Since the cost of replicating the request itself is negligible, with k members available to play the coordinator role for distinct requests, users obtained a k-fold speedup. Moreover, because the client's connection to the group wouldn't break even if a fault did occur, the client was completely insulated from failures. The mechanism was very popular.
- Many applications adopted a publish-subscribe communication pattern. As mentioned above, Isis offered a "news" interface that supported what later became known as topic-based publish-subscribe. In the simplest model, each topic maps one-to-one to a process group, but this creates huge numbers of groups. Accordingly, the tool used a form of channelization, mapping each topic to one of a small set of groups and then filtering incoming messages to deal with the resulting inaccuracy. This approach remains common in modern publish-subscribe products.

With the exception of publish-subscribe applications, it is interesting to realize that most uses of Isis involved servers running on small clusters. For example, the French Air Traffic Control System runs Isis in datacenters with hundreds of machines, but

[9] In Isis, a multicast could be invoked asynchronously (no replies), or could wait for replies from one, several, or all group members.

organized as clusters of 3 to 5 consoles. Isis was configured to run in disjoint configurations, keeping loads light and providing fault isolation.

Publish-subscribe, however, is a very different world: data rates can be very high, groups can be big, and enterprises may have other desires too, such as security or management interfaces. Group communication of all kinds, not merely virtual synchrony, is challenged by such goals — indeed, operators of today's largest datacenter platforms report that instability of large-scale publish-subscribe deployments represents a very serious problem, and we know of a number of very high-profile settings in which publish-subscribe has effectively been banned because the technology proved to be unreliable at high data rates in large-scale uses. Such stories make it clear that the French Air Traffic Control project made a very wise decision. Later, we'll comment on our recent work to overcome these issues, but they clearly point to important research challenges.

6.8.2 Causal and Other Controversies

Although virtual synchrony has certainly been successful and entered the mainstream computing world, this history wouldn't be complete without at least allusion to some of the controversies mentioned earlier. There were many of them:

- The causally ordered multicast primitive used in Isis was debated with enormous enthusiasm (and much confusion) [23, 9].
- There was a period of debate about the applicability of the FLP result. The question was resolved emphatically with the not-surprising finding that indeed, consensus and virtual synchrony are related [25, 53, 67].
- We noted that the formal definition of consensus includes an agreement property that Isis violates by default. Is virtual synchrony therefore incorrect by default?
- Because Paxos can be used to build multicast infrastructures, and virtual synchrony communication systems can be used to solve consensus, one can ask which is "better". Earlier, we noted that virtual synchrony can implement guarantees identical to Paxos if the user limits himself to uniform multicast or *gbcast*, or uses *flush*. As noted earlier, systems like Chubby do use Paxos, but tend to be engineered with all sorts of optimizations and additional mechanisms: Paxos is just one of several protocols, just as the virtual synchrony view update protocol is just one of many Isis protocols. Thus, it makes little sense to talk about choosing "between" Paxos and virtual synchrony. The protocol suites we end up using incorporate elements of both.
- There was much interest in using groups to securely replicate keys for purposes of end-to-end cryptographic security. Interestingly, this model runs afoul of the cloud-computing trend towards hosting everything: these days, companies like Google want to manage our medical records, provide transcripts of telephone calls, and track our digital lives. Clearly, one is supposed to trust one's cloud provider, and perhaps for this reason, the major security standards are all client-server infrastructures; true end-to-end security keys that might deny the cloud platform a chance to see the data exchanged among clients have no obvious role. But this could change, and if so, secure group keys could be just what the doctor ordered.

6.8.3 What Next? Live Objects and Quicksilver Scalable Multicast!

The story hasn't ended. Today's challenges relate to scale and embeddings. With respect to scalability, the push towards cloud computing has created a new interest on infrastructure for datacenter developers. The tools used in such settings must scale to accommodate deployments on tens or hundreds of thousands of machines and correspondingly high data rates. Meanwhile, out at the edge, replication and multicast patterns are increasingly interesting in support of new forms of collaboration and new kinds of social networking technologies.

At Cornell, our current focus is on solving these next generation scalability challenges, while also integrating reliable multicast mechanisms with the modern generation of componentized platforms that support web services standards — for example, the Microsoft .net platform and the J2EE platform favored by Java developers. We've created a system that implements what we are calling "Live Distributed Objects[10] " [60, 62]. The basic idea is to enable end-users, who may not be programmers, to build applications by drag-and-drop, much as one pulls a figure or a table into a text document.

From the perspective of the application designer, live objects are edge-mashups, created on the client platform much in the same sense as a Google mashup that superimposes push-pin locations on maps: the user drags and drops objects, constructing a graph of components that interact by event passing. The main difference is that the Google mashup is created on Google's platform and exported through a fairly sophisticated minibrowser with zoom, pan and layer controls; a live object is a simpler component designed to connect with other live objects within the client machine to form a graph that might have similar functionality to the Google version, but could import content from multiple hosted platforms (for example, we routinely combine Google maps with Yahoo! weather and population data from the National Census), and with peer-to-peer protocols that can achieve very low latency and jitter when clients communicate with one-another. Once created, a Live Object-based application can be shared by making copies — it can even be emailed — and each node that activates it will effectively become an endpoint of a group associated with that object.

We've packaged a number of multicast protocols as Live Objects, and this creates a connection to the theme of the present article: one of the protocols supports virtually synchronous replication at high data rates and large scale. However, not all objects have complex distributed behaviors. Live objects can also be connected to sensors, actuators, applications that generate events, and even databases or spreadsheets.

With Live Objects, we're finding that even an unskilled user can build non-trivial distributed collaboration applications, workflow systems, or even games. The experience is very similar to building scenarios in games like Second Life, but whereas Second Life "runs" on a data center, Live Objects run directly on and between the client platforms where the live application is replicated. Although doing so poses

[10] A video of a demo can be seen at http://liveobjects.cs.cornell.edu

many challenges, one of our research goals is to support a version of Second Life built entirely with Live Objects.

In support of Live Objects, we've had to revisit reliable multicast and replication protocols [61]. As noted, existing solutions can scale a single group to perhaps 100 members, but larger groups tend to destabilize at high data rates. None of the systems we've evaluated can handle huge numbers of groups with irregular overlap. Yet, even simple Live Object applications can create patterns of object use in which a single machine might end up joining thousands of replication groups, and extremely high data rates. In [60] we discuss some of the mechanisms we're exploring in support of these new dimensions of scalability. With these, we believe that groups providing a slightly weaker reliability model than virtual synchrony can scale to at least hundreds of members, can sustain data rates as high as 10,000 1-kbyte messages per second, and individual nodes can join thousands of groups that overlap in irregular ways.

We're also revisiting the way that virtual synchrony, consensus and transactional guarantees are implemented. The standard way to build such protocols is to do so as a library constructed directly over UDP message passing. We're currently working on a scripting language (we call it the *properties framework*) in which higher level reliability properties can be described. An interpretive runtime executes these scripts in a scalable, asynchronous, dataflow manner. Preliminary results suggest that strong reliability properties can scale better than had previously been believed, but we'll need to complete the work to know for sure.

Live objects include a simple type system, matched to the limited interface model favored in modern web services platforms, but far from the state of the art. Readers interested in connections between replication and type theory may want to look at papers such as [43, 44, 47, 51]. Research on componentized protocols includes [6, 7, 37, 39, 56, 73]. These lines of study come together in work on typed endpoints in object oriented systems, such as [18, 28, 32, 34, 33].

6.9 Closing Thoughts

It seems appropriate to end by sharing an observation made by Jim Gray, who (over dinner at a Microsoft workshop) commented on a parallel between the early database community, and what he believed has happened with virtual synchrony and other strong replication models. In its early days, the transactional community aggressively embraced diversity. Researchers published on all sorts of niche applications and papers commonly argued for specialized variations on the transactional model. The field was awash in specialized database systems. Yet real success only came with consolidation around transactions on relational databases: so much investment was focused on the model that the associated technology advanced enormously.

With this success, some researchers probably felt that the field was taking a step "backwards", abandoning superior solutions in favor of less elegant or less efficient ones. Yet success also brought research opportunities: research was needed to over-

come a new set of challenges of scale, and performance. The science that emerged was no less profound than the science that had been "displaced."

In this, Jim saw a general principle. If a technology tries too hard to make every user happy, so much effort is needed to satisfy the 20% with the hardest problems that the system ends up being clumsy and slow. The typical user won't need most of its features, and many will opt for a simpler, cheaper solution that's easier to use. The irony is that in striving to make every user happy, a technology can actually leave the majority *unhappy*. In the end, an overly ambitious technology merely marginalizes itself.

Did the Isis system actually "need" four flavors of ordered multicast? Probably not: we got carried away, and it made the system difficult for the community to understand.

Today, the opportunity exists to create consistency-preserving replication tools that might be widely adopted, provided that we focus on making replication as easy as possible to use in widely standard platforms. In some ways this may force us to focus on a least common denominator approach to our past work. Yet making replication with strong semantics work for real users, on the scale of the Internet, also reveals profound new challenges, and as we solve them, we may well discover that the underlying science is every bit as interesting and deep as anything we discovered in the past.

Acknowledgements The author is grateful to André Schiper, Robbert van Renesse and Fred Schneider. Not only would virtual synchrony not exist in its current form without the efforts of all three, but they were generous enough to agree to read an early draft of this paper. This revision is certainly greatly improved as a result.

References

1. Aguilera, M., Merchant, A., Shah, M., Veitch, A., Karamanolis, C.: Sinfonia: a new paradigm for building scalable distributed systems. In: 21st ACM SOSP, Nov. 2007, pp. 159–174 (2007)
2. Amir, Y., Nita-Rotaru, C., Stanton, J., Tsudik, G.: Secure Spread: An Integrated Architecture for Secure Group Communication. IEEE TDSC 2(3) (2005)
3. Babaoglu, Ö., Bartoli, A., Dini, G.: Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. IEEE Transactions on Computers 46(6), 642–658 (1997)
4. Bernstein, P., Goodman, N.: Concurrency Control in Distributed Database Systems. ACM Computing Surveys 13(2) (1981)
5. Bernstein, P., Goodman, N.: An algorithm for concurrency control and recovery in replicated distributed databases. ACM Transactions on Database Systems 9(4), 596–615 (1984)
6. Bhatti, N., Hiltunen, M., Schlichting, R., Chiu, W.: Coyote: A System for Constructing Fine-Grain Configurable Communication Services. ACM Transactions on Computer Systems 16(4), 321–366 (1998)
7. Biagioni, E., Harper, R., Lee, P.: A Network Protocol Stack in Standard ML. Journal of Higher-Order and Symbolic Computation 14(4) (2001)
8. Birman, K.: Replication and Fault-Tolerance in the ISIS System. In: 10th ACM Symposium on Operating Systems Principles, Dec. 1985, pp. 79–86 (1985)
9. Birman, K.: Responses to Cheriton and Skeen's SOSP paper on Understanding the Limitations of Causal and Total Event Ordering. SIGOPS Operating Systems Review 28(1) (1994)

10. Birman, K.: A review of experiences with reliable multicast. Software Practice and Experience 29(9) (1999)
11. Birman, K.: Reliable Distributed Systems. Springer, New York (2004)
12. Birman, K., Chandersekaran, C., Dolev, D., Van Renesse, R.: How the Hidden Hand Shapes the Market for Software Reliability. In: Proceedings IEEE Workshop on Applied Software Reliability, Philadelphia, PA (June 2006)
13. Birman, K., Joseph, T.: Reliable communication in the presence of failures. Tech. Rep. TR85-694 (August 1985)
14. Birman, K., Joseph, T.: Exploiting Virtual Synchrony in Distributed Systems. In: 11th ACM Symposium on Operating Systems Principles (Dec. 1987)
15. Birman, K., Joseph, T.: Reliable communication in the presence of failures. ACM Transactions on Computer Systems 5(1) (1987)
16. Birman, K., Joseph, T., Raeuchle, T., El Abbadi, A.: Implementing Fault-Tolerant Distributed Objects. IEEE Transactions on Software Engineering 11(6) (1985)
17. Birman, K., Schiper, A., Stephenson, P.: Lightweight causal and atomic group multicast. ACM Transactions on Computer Systems 9(3), 272–314 (1991)
18. Briot, J., Guerraoui, R., Lohr, K.: Concurrency and Distribution in Object-Oriented Programming. ACM Comput. Surv. 30(3), 291–329 (1998)
19. Burrows, M.: The Chubby Lock Service for Loosely-Coupled Distributed Systems. In: OSDI, pp. 335–350 (2006)
20. Chandra, T., Hadzilacos, V., Toueg, S., Charron-Bost, B.: On the impossibility of group membership. In: Proc. 15th PODC, May 23-26, 1996, pp. 322–330 (1996)
21. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos Made Live — An Engineering Perspective (based on Chandra's 2006 invited talk). In: Proc. 26th PODC, Aug. 2007, pp. 398–407 (2007)
22. Chang, J., Maxemchuk, N.: Reliable broadcast protocols. ACM Trans. on. Computer Systems 2(3), 251–273 (1984)
23. Cheriton, D., Skeen, D.: Understanding the Limitations of Causally and Totally Ordered Communication. In: SOSP, pp. 44–57 (1993)
24. Cheriton, D., Zwaenepoel, W.: Distributed process groups in the V Kernel. ACM Transactions on Computer Systems (TOCS) 3(2), 77–107 (1985)
25. Chockler, G., Keidar, I., Vitenberg, R.: Group Communication Specifications: A Comprehensive Study. ACM Computing Surveys 33(4) (2001)
26. Chrysanthis, P.K., Ramamritham, K.: ACTA: the SAGA Continues. In: Elmagarmid, A. (ed.) Database transaction models for advanced applications, Morgan Kaufmann, San Francisco (1992)
27. Cristian, F., Aghili, H., Strong, R., Volev, D.: Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In: Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15), Ann Arbor, MI, USA, June 1985, pp. 200–206. IEEE Computer Society Press, Los Alamitos (1985)
28. Damm, C., Eugster, P., Guerraoui, R.: Linguistic Support for Distributed Programming Abstractions. In: CDCS, pp. 244–251 (2004)
29. Dekel, E., et al.: Distribution and Consistency Services (DCS), `http://www.haifa.ibm.com/projects/systems/dcs/index.html`
30. Dekel, E., Frenkel, O., Goft, G., Moatti, Y.: Easy: engineering high availability QoS in wservices. In: Proc. 22nd Reliable Distributed Systems, pp. 157–166 (2003)
31. Dolev, D., Malkhi, D.: The Transis Approach to High Availability Cluster Communication. Comm. ACM 39(4), 87–92 (1996)
32. Eugster, P.: Type-based Publish/Subscribe: Concepts and Experiences. ACM Transactions on Programming Languages and Systems (TOPLAS) 29(1) (2007)
33. Eugster, P., Damm, C., Guerraoui, R.: Towards Safe Distributed Application Development. In: ICSE, pp. 347–356 (2004)
34. Eugster, P., Guerraoui, R., Damm, C.: On Objects and Events. In: OOPSLA, pp. 254–269 (2001)

35. Fischer, M., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process (initially published in ACM PODS, August 1983). Journal of the ACM (JACM) 32(2) (1985)
36. Guerraoui, R., Schiper, A.: Consensus Service: A Modular Approach for Building Agreement Protocols in Distributed Systems. In: Proc. 26th FTCS, Japan, June 1996, pp. 168–177 (1996)
37. Hayden, M.: The Ensemble System. Ph.D. thesis, Cornell University, available as TR 98-1662 (May 1998)
38. Herlihy, M., Wing, J.: Linearizability: A Correctness Condition for Concurrent Objects. ACM TOPLAS 12(3), 463–492 (1990)
39. Hutchinson, N.C., Peterson, L.L.: The x-Kernel: An architecture for implementing network protocols. IEEE Trans. Software Eng. 17(1) (1991)
40. Joseph, T.A., Birman, K.: Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems. ACM Trans. Comput. Syst. 4(1), 54–70 (1986)
41. Kaashoek, M.F., Tanenbaum, A.S., Verstoep, K.: Group Communication in Amoeba and its Applications. Distributed Systems Engineering 1(1), 48–58 (1993)
42. Keidar, I., Dolev, D.: Increasing the Resilience of Atomic Commit at no Additional Cost. In: ACM PODS, May 1995, pp. 245–254 (1995)
43. Keidar, I., Khazan, R., Lynch, N., Shvartsman, A.: An Inheritance-Based Technique for Building Simulation Proofs Incrementally. ACM TOSEM 11(1) (2002)
44. Krumvieda, C.: Distributed ML: Abstractions for Efficient and Fault-Tolerant Programming. Ph.D. thesis, Cornell University, available as TR 93-1376 (1993)
45. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Comm. ACM 21(7) (1978)
46. Lamport, L.: Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. ACM TOPLAS 6(2) (1984)
47. Lamport, L.: The temporal logic of actions. ACM TOPLAS 16(3), 872–923 (1994)
48. Lamport, L.: The Part-Time Parliament (technical report version: 1990). ACM Transactions on Computer Systems 16(2), 133–169 (1998)
49. Liskov, B., Scheifler, R.: Guardians and Actions: Linguistic Support for Robust, Distributed Programs. ACM TOPLAS 5(3) (1983)
50. Liu, X., Kreitz, C., van Renesse, R., Hickey, J., Hayden, M., Birman, K., Constable, R.: Building reliable, high-performance communication systems from components. In: 17th ACM SOSP (Dec. 1999)
51. Lynch, N., Tuttle, M.: An Introduction to Input/Output automata (also Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology). CWI Quarterly 2(3), 219–246 (1989)
52. Malloth, C.P., Felber, P., Schiper, A., Wilhelm, U.: Phoenix: A Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale. In: Proc. of IEEE Workshop on Parallel and Distributed Platforms in Industrial Products, San Antonio, TX (Oct. 1995)
53. Malloth, C.P., Schiper, A.: View Synchronous Communication in the Internet. Tech. Rep. 94/84, EPFL (Oct. 1994)
54. Manferdelli, J.: Microsoft Corporation. Unpublished correspondence (Oct. 2007)
55. McKendry, M.S.: Clouds: A fault-tolerant distributed operating system. IEEE Tech. Com. Distributed Processing Newsletter 2(6) (1984)
56. Mishra, S., Peterson, L.L., Schlichting, R.D.: Experience with modularity in Consul. Software—Practice and Experience 23(10) (1993)
57. Moser, L.E., Amir, Y., Melliar-Smith, P.M., Agarwal, D.A.: Extended virtual synchrony. In: Proceedings of the 14th IEEE International Conference on Distributed Computing Systems, Poznan, Poland, June 1994, pp. 56–65 (1994)
58. Moser, L.E., Melliar-Smith, P.M., Agarwal, D., Budhia, R.K., Lingley-Papadopoulos, C.A., Archambault, T.: The Totem system. In: Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing, Pasadena, CA (June 1995)
59. Moser, L.E., Melliar-Smith, P.M., Narasimhan, P.: The Eternal System. In: Workshop on Dependable Distributed Object Systems, OOPSLA'97, Atlanta, Georgia (October 1997)

60. Ostrowski, K., Birman, K., Dolev, D.: Live Distributed Objects: Enabling the Active Web. IEEE Internet Computing (Nov./Dec. 2007)
61. Ostrowski, K., Birman, K., Dolev, D.: QuickSilver Scalable Multicast. In: Network Computing and Applications (NCA), Cambridge, MA (2008)
62. Ostrowski, K., Birman, K., Dolev, D., Ahnn, J.H.: Programming with live distributed objects. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 463–489. Springer, Heidelberg (2008)
63. Randell, B., Xu, J.: The Evolution of the Recovery Block Concept. In: Lyu, M.R. (ed.) Software Fault Tolerance, pp. 1–21. John Wiley & Sons, Chichester (1995)
64. Reed, B., Junqueira, F., Konar, M.: Zookeeper: Because Building Distributed Systems is a Zoo. Submitted for publication (Oct. 2007)
65. Ricciardi, A., Birman, K.: Using Process Groups to Implement Failure Detection in Asynchronous Environments. In: PODC, pp. 341–353 (1991)
66. Schiper, A., Sandoz, A.: Uniform reliable multicast in a Virtually Synchronous Environment. In: Proc. 13th ICDCS, Pittsburgh (May 1993)
67. Schiper, A., Sandoz, A.: Primary Partition "Virtually-Synchronous Communication" Harder than Consensus. In: Tel, G., Vitányi, P.M.B. (eds.) WDAG 1994. LNCS, vol. 857, pp. 39–52. Springer, Heidelberg (1994)
68. Schneider, F.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys 22(4), 299–319 (1990)
69. Schneider, F., Schlichting, R.: Fail-stop processors: An approach to designing fault-tolerant computing systems. TOCS 1(3), 222–238 (1983)
70. Skeen, D.: Nonblocking Commit Protocols. In: Proc. ACM SIGMOD, pp. 133–142 (1981)
71. Skeen, D.: Determining the Last Process to Fail. In: ACM PODS, pp. 16–24 (1983)
72. Van Renesse, R., Birman, K.: Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press, Los Alamitos (1994)
73. Van Renesse, R., Birman, K., Maffeis, S.: Horus: A Flexible Group Communication System. Communications of the ACM 39(4), special issue on Group Communication Systems (1996)