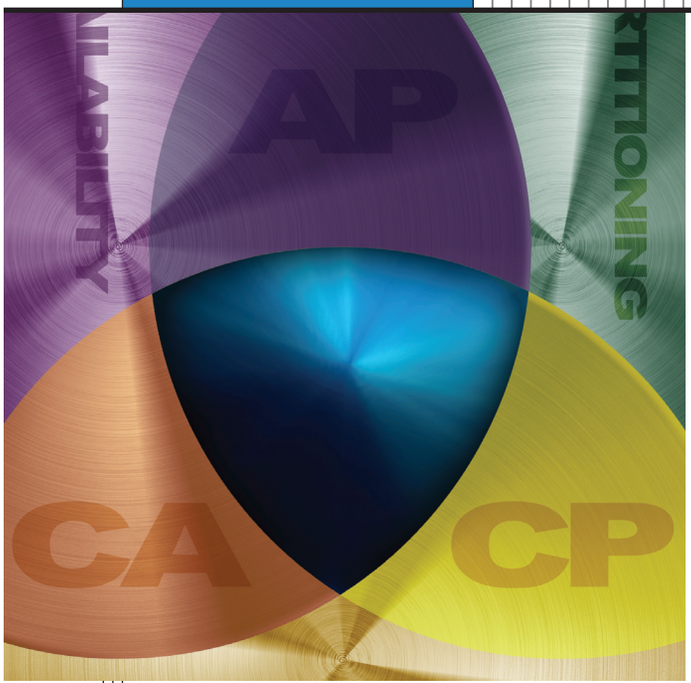


Overcoming CAP with Consistent Soft-State Replication

Kenneth P. Birman, Daniel A. Freedman, Qi Huang, and Patrick Dowell, *Cornell University*

New data-consistency models make it possible for cloud computing developers to replicate soft state without encountering the limitations associated with the CAP theorem.

The CAP theorem explores tradeoffs between *consistency*, *availability*, and *partition tolerance*, and concludes that a replicated service can have just two of these three properties.[1,2] To prove CAP, researchers construct a scenario in which a replicated service is forced to respond to conflicting requests during a wide-area network outage, as might occur if two different datacenters hosted replicas of some single service, and received updates at a time when the network link between them was down. The replicas respond without discovering the conflict, resulting in inconsistency that might confuse an end user.

However, there are important situations in which cloud computing developers depend upon data or service replication, and for which this particular proof does not seem to apply. Here, we consider such a case: a scalable service running in the first tier of a single datacenter. Today's datacenters employ redundant networks that almost never experience partitioning failures: the "P" in CAP does not occur. Nonetheless, many cloud computing application developers believe in a generalized CAP "folk theorem," holding that scalability and elasticity are incompatible with strong forms of consistency.

Our work explores a new consistency model for data replication in first-tier cloud services. The model combines agreement on update ordering with a form of durability that we call *amnesia freedom*. Our experiments confirm that this approach scales and performs surprisingly well.

THE CAP THEOREM'S BROAD REACH

The CAP theorem has been highly influential within the cloud computing community, and is widely cited as a justification for building cloud services with weak consistency or assurance properties. CAP's impact has been especially important in the first-tier settings on which we focus in this article.

Many of today's developers believe that CAP precludes consistency in first-tier services. For example, eBay has proposed BASE (Basically Available replicated Soft state with Eventual consistency), a development methodology in which services that run in a single datacenter on a reliable network are deliberately engineered to use potentially stale or incorrect data, rejecting synchronization in favor of faster response, but running the risk of inconsistencies.[3] Researchers at Amazon.com have also adopted BASE. They point to the self-repair mechanisms in the Dynamo key-value store as an example of how eventual consistency behaves in practice.[4]

Inconsistencies that occur in eBay and Amazon cloud applications can often be masked so that users will not notice them. The same can be said for many of today's most popular cloud computing uses: how much consistency is really needed by YouTube or to support Web searches? However, as applications with stronger assurance needs

 Published by the IEEE Computer Society 0018-9162/12/$31.00 © 2012 IEEE

r2bir.indd 50 1/25/12 11:36 AM

migrate to the cloud, even minor inconsistencies could endanger users. For example, there has been considerable interest in creating cloud computing solutions for medical records management or control of the electric power grid. Does CAP represent a barrier to building such applications, or can stronger properties be achieved in the cloud?

At first glance, it might seem obvious that the cloud can provide consistency. Many cloud applications and products offer strong consistency guarantees, including databases and scalable global file systems. But these products do not run in the first tier of the cloud—the client-facing layer that handles incoming requests from browsers or responds to Web services method invocations. There is a perception that strong consistency is not feasible in these kinds of highly elastic and scalable services, which soak up much of the workload.

Here, we consider first-tier applications that replicate data—either directly, through a library that offers a key-value storage API, or with some form of caching. We ignore application details, and instead look closely at the multicast protocols used to update replicated data. Our work confirms that full-fledged atomic multicast probably does not scale well enough for use in this setting (a finding that rules out using durable versions of Paxos or ACID transactions to replicate data), and in this sense we agree with the generalization of CAP. However, we also advocate other consistency options that scale far better: the amnesia-free protocols. By using them, developers can overcome the limitations associated with CAP.

THE FIRST-TIER PROGRAMMING MODEL

Cloud computing systems are generally structured into tiers: a first tier that handles incoming client requests (from browsers, applications using Web services standards, and so on), caches and key-value stores that run near the first tier, and inner-tier services that provide database and file system functionality. Back-end applications work offline, preparing indices and other data for later use by online services.

Developers typically tune applications running in the first tier for rapid response, elasticity, and scalability by exploiting a mixture of aggressive replication and very loose coupling between replicas. A load-balancer directs requests from client systems to a service instance, which runs the developer-provided logic. To minimize delay, an instance will compute as locally as possible, using cached data if available, and launching updates in the background (asynchronously).

In support of this model, as little synchronization is done as possible: first-tier services are nontransactional and run optimistically, without obtaining locks or checking that cached data is still valid. Updates are applied optimistically to cached data, but the definitive outcome will not occur until the real update is finally done later, on the underlying data; if this yields a different result, the system will often simply ignore the inconsistency. While first-tier services can query inner services, rather than focusing on that pattern, we consider reads and updates that occur entirely within the first tier, either on data replicated by first-tier service instances or within the key-value stores and caches supporting them.

Modern cloud development platforms standardize this model, and in fact take it even further. First-tier applications are also required to be stateless: to support rapid instance launch or shutdown, cloud platforms launch each new instance in a standard initial state, and discard any local data when an instance fails or is halted by the management infrastructure. "Stateless" doesn't mean that these instances have no local data at all, but rather that they are limited to a *soft state* that won't be retained across instance failure/launch sequences. On launch, such state is always clean; a new instance initializes itself by copying data from some operational instance or by querying services residing deeper in the cloud.

Cloud computing systems are generally structured into tiers: a first tier that handles incoming client requests, caches and key-value stores that run near the first tier, and inner-tier services that provide database and file system functionality.

Even an elastic service will not be completely shut down without warning: to ensure continuous availability, cloud management platforms can be told to keep some minimum number of replicas of each service running. Thus, these services vary the number of *extra* replicas for elasticity, but preserve a basic level of availability.

Jointly, these observations enable a form of weak durability. Data replicated within the soft state of a service, in members that the management platform will not shut down (because they reside within the core replica set), will remain available unless a serious failure causes all the replicas to crash simultaneously, a rare occurrence in a well-designed application.

Today's first-tier applications often use stale or otherwise inconsistent data when responding to requests because the delays associated with fetching (or even validating) data are perceived as too high. Even slight delays can drive users away. Web applications try to hide resulting problems or to clean them up later (*eventual consistency*), possibly ending up in a state at odds with what the client saw. Cloud owners justify these architectural decisions by asserting a generalized CAP principle, arguing that consistency is simply incompatible with scalability and rapid responsiveness.

r2bir.indd 51 1/25/12 11:36 AM

Adopting a consistency model better matched to first-tier characteristics permits similar responsiveness while delivering stronger consistency guarantees. This approach could enable cloud-hosting of applications that need to justify the responses they provide to users, such as medical systems that monitor patients and control devices. Consistency can also enhance security: a security system that bases authorization decisions on potentially stale or incorrect underlying data is at risk of mistakes that a system using consistent data will not make.

CONSISTENCY: A MULTIDIMENSIONAL PROPERTY

Developers can define consistency in many ways. Prior work on CAP defined "C" by citing the ACID (atomicity, consistency, isolation, and durability) database model; in this approach the "C" in CAP is defined to be the "C" and "D" from ACID. Consistency, in effect, is conflated with durability. Underscoring this point, several CAP and BASE articles cite Paxos as the usual way to implement consistent replication: an atomic multicast protocol that provides total ordering and durability.

A security system that bases authorization decisions on potentially stale or incorrect underlying data is at risk of mistakes that a system using consistent data will not make.

Durability is the guarantee that once an update has been ordered and marked as deliverable, it will never be lost, even if the entire service crashes and then restarts. But for a first-tier service, durability in this direct sense is clearly not feasible: any state that can endure such a failure would necessarily be stored outside the service itself. Thus, by focusing on mechanisms that expend effort to guarantee durability, CAP researchers arrive at conclusions that are not applicable in first-tier services, which are limited to soft state.

Membership

Any replication scheme needs a *membership model*. Consider some piece of replicated data in a first-tier service: the data might be replicated across the full set of first-tier application instances or it might reside just within some small subset of them (in the latter case the term *shard* is often used). Which nodes are supposed to participate?

For the case in which every replica has a copy of the data item, the answer is evident: all the replicas currently running. But because cloud platforms vary this set elastically, the actual collection will change over time, perhaps rapidly. Full replication necessitates tracking the set, having a policy for initializing a newly launched service instance, and ensuring that each update reaches all the replicas, even if that set is quite large.

For sharded data, any given item will be replicated at just a few members, hence a mapping from key (item-id) to shard is needed. Since each service instance belongs to just a few shards but potentially needs access to all of them, a mechanism is also needed so that any instance can issue read or update requests to any shard. Moreover, since shard membership will vary elastically (but without completely shutting down the full membership of any shard), membership dynamics must be factored into the model.

One way to handle such issues is seen in Amazon's Dynamo key-value store,[5] which is a form of distributed hash table (DHT). Each node in Dynamo is mapped (using a hashing function) to a location on a virtual ring, and the key associated with each item is similarly mapped to the ring. The closest node with a mapped id less than or equal to that of the item is designated as its primary owner, and the value is replicated to the primary and to the next few (typically three) nodes along the ring: the shard for that key. Shard mappings change as nodes join and leave the ring, and data is moved around accordingly (a form of *state transfer*). Amazon apparently coordinates this with the cloud management service to minimize abrupt elasticity decisions that would shut down entire shards faster than the members can transfer state to new owners.

A second way to implement shards occurs in systems that work with *process groups*:[6] here, a group communication infrastructure such as our new Isis[2] system solves the various requirements. Systems of this sort offer an API that provides much of the needed functionality: ways for processes to create, join, and leave groups; group names that might encode a key (such as "shard123"); a state transfer mechanism to initialize a joining member from the state of members already active; and other synchronization features. Isis[2], for example, implements the virtual synchrony model.[7] The developer decides how shards should work, then uses the provided API to implement the desired policy over the group infrastructure tools. Again, coordination with the cloud management infrastructure is required to avoid disruptive elasticity events.

Services that run on a stable set of nodes for a long period employ a third shard-implementation option that enables a kind of *static* membership in which some set of nodes is designated as running the service. Here, membership remains fixed, but some nodes might be down when a request is issued. This forces the use of quorum replication schemes, in which only a quorum of replicas see each update, but reading data requires accessing multiple replicas; state transfer is not needed unless the static membership must be reconfigured—a rare and costly operation. Several CAP articles refer to the high cost of quorum operations, suggesting that many in the

r2bir.indd 52 1/25/12 11:36 AM

community have had direct experience with this category of solutions, which includes the most widely used Paxos libraries.

Notice that quorum operations are needed in this static membership case, but not in the others: the DHT and process group approaches avoid the need for quorums because they evolve shard membership as nodes join, leave, or fail. This matters because quorum operations residing on the critical path for a first-tier request introduce nonlocal delays, which cloud developers are intent upon minimizing in the interests of rapid response and scalability. With dynamic shard membership both reads and writes can be done *locally*. The multicasts that update remote replicas occur asynchronously, in parallel with computation of the response that will be sent to the client.

Update ordering

A second dimension of consistency concerns the policy for applying updates to replicas. A consistent replication scheme applies the same updates to every replica in the same order and specifies the correct way to initialize new members or nodes recovering from a failure.[6-8]

Update ordering costs depend on the pattern for issuing updates. In many systems, each data item has a primary copy through which updates are routed, and one or more replicas function as hot-standbys and can support read-only operations. Some systems shift the role of being primary around, but the basic idea is the same: in both cases, delivering updates in the order they were sent, without gaps, keeps the system consistent across multiple replicas. The resulting problem is very much like FIFO delivery of data with TCP, and is solved in much the same manner.

A more costly multicast ordering property is needed if every replica can initiate concurrent, conflicting updates to the same data items. When concurrent updates are permitted, the multicast mechanism must select an agreed-upon order, then the delivery order ensures that the replicas apply the updates in a consistent order. The numerous ways to build such mechanisms are not tremendously complex, but they do introduce extra protocol steps, which slows down the update protocol. The CAP and BASE literature does not discuss this issue explicitly, but the examples used point to systems that permit concurrent updates. Simply requiring replicated data to have a primary copy can yield a significant cost reduction.

Durability

Yet a third dimension involves durability of updates. Obviously, an update that has been performed is durable if the service will not forget it. But precisely what does it mean to have "performed" an update? Must the durability mechanism retain data across complete shutdowns of a service or shard's full membership?

In applications where the goal is to replicate a database or file (some form of external storage), durability involves mechanisms such as write-ahead logs: all the replicas push updates to their respective logs, then acknowledge that they are ready to commit the update; in a second phase, the system can apply the updates in the logs to the actual database. Thus, while the Paxos protocol[9,10] does not refer to the application per se, most Paxos implementations are designed to work with external applications like databases for which durability entails logging of pending updates. This presumes durable storage that will survive failures. The analogous database mechanism would be a write-ahead log.

But first-tier services are required to be stateless. Can they replicate data in a way that offers a meaningful durability property? The obvious answer is to consider in-memory update replication: we could distinguish between a service that might respond to a client *before* every replica knows of the updates triggered by that client's request, and a service that delays until *after* every replica has acknowledged the relevant updates. We call the former solution *nondurable*: if the service has n members, even a single failure can leave $n - 1$ replicas in a state where they will never see the update. We refer to the latter solution as amnesia freedom: the service will not forget the update unless all n members fail. Recall that coordinating with the cloud management platform minimizes abrupt shutdowns of all n replicas.

Simply requiring replicated data to have a primary copy can yield a significant cost reduction.

Amnesia freedom is not perfect. If a serious failure forces an entire service or shard to shut down, unless the associated data is backed up on an inner-tier service, state will be lost. But because this is a rare case, such a risk may be quite acceptable. For example, suppose that applications for monitoring and controlling embedded systems such as medical monitoring devices move to cloud-hosted settings. While these applications require consistency and other assurance properties, the role of online monitoring is continuous. Moreover, applications of this sort generally revert to a fail-safe mode when active control is lost. Thus, an inner-tier service might not be needed.

Applications that do push updates to an inner service have a choice: they could wait for the update to be acknowledged or they could adopt amnesia freedom. However, in so doing, they accept a window of vulnerability for the (hopefully brief) period after the update is fully replicated in the memory of the first-tier service, but before it reaches the inner tier.

r2bir.indd 53 1/25/12 11:36 AM

Sometimes, a rapid response might be so important that it is not possible to wait for an inner-tier response; if so, amnesia freedom represents a compromise that greatly reduces the risk of update loss at very low cost. To offer another analogy, while database products generally support true multicopy serializability, costs can be high. For this reason, database mirroring is more often done by asynchronously streaming a log, despite the small risk that a failure could cause updates to be lost. Thus, databases often operate in the same manner as an amnesia-free solution that asynchronously pushes updates to an inner-tier service.

Failure mode

Our work assumes that applications fail by crashing and that network packets can be lost. Within a single datacenter, if a partition occurs, the affected machines are treated as if they had crashed: when the partition is repaired, those nodes will be forced to restart.

Fusing order-based consistency with amnesia freedom results in a strongly consistent model with a slightly weaker notion of durability.

Putting it all together

From this complex set of choices and options, it is possible to construct diverse replication solutions with very different properties, required structure, and expected performance. While some make little sense in the first tier; others represent reasonable options, including the following:

- Build protocols that replicate data optimistically and later heal any problems that arise, perhaps using gossip (BASE). Updates are applied in the first tier, but then passed to inner-tier services that might perform them in different orders.
- Build protocols synchronized with respect to membership changes, and with a variety of ordering and durability properties—virtual synchrony and also "in-memory" versions of Paxos, where the Paxos durability guarantee applies only to in-memory data. Simply enforcing a barrier can achieve amnesia freedom: the system pauses if needed, delaying the response until any updates initiated by the request (or seen by the request through its reads) have reached all the replicas and thus become stable.
- Implement the state machine replication model, including strong durability—the guarantee that even if all service members crash, the current state will be recoverable. Most Paxos implementations use this model. However, in our target scenario, strong durability is not meaningful, and the first phase is limited to logging messages in the memory of the replicas themselves.
- Implement database transactions in the first tier, coupling them to the serialization order using inner tiers, for example, via a true multicopy model based on the ACID model or a snapshot-isolation model. The research community is currently investigating this approach.

How does CAP deal with these diverse options? The question is easier to pose than to answer. When Eric Brewer proposed CAP as a general statement,[1] he offered the classic partitioning failure scenario as an illustration of a more broadly applicable principle. His references to consistency evoked ACID database properties. Seth Gilbert and Nancy A. Lynch offered their proof in settings with partitionable wide-area links.[2] They pointed out that with even slight changes, CAP ceases to apply, and proposed a *t*-eventual consistency model that avoids the CAP tradeoff.

In their work on BASE, Dan Pritchett[3] and Werner Vogels[4] pointed to both the ACID model and the durable form of Paxos.[9,10] They argued that these models are too slow for use in the first tier, expressing concerns that apparently stem from the costly two-phase structure of these particular protocols, and their use of quorum reads and updates, resulting in nonlocal responsiveness delays on the critical path that computes responses.

The concerns about performance and scalability relate to durability mechanisms, not order-based consistency. Because sharded data predominates in the first tier, it is possible to require each shard to distinguish a primary copy where updates are performed first. Other replicas mirror the primary, supporting read-accesses, and taking over as primary only in the event of a reconfiguration. For a DHT-like structure, it is possible to use chain replication;[11] in a process group, the system would load-balance the reads while routing updates through the primary, which would issue the needed multicasts. This reduces the cost of consistency to the trivial requirement of performing updates in FIFO order. Of course, this also requires synchronizing group membership changes relative to updates, but there is no reason that this should impact steady-state performance.

We favor in-memory logging of updates, leading to amnesia freedom. Fusing order-based consistency with amnesia freedom results in a strongly consistent model with a slightly weaker notion of durability. Indeed, when acting on a request from an external user, any replica can perform everything locally with the appropriate data, up to the very last step: before responding to the external client, any updates triggered by a request must be stable—that is, they must have reached all the relevant replicas.

r2bir.indd 54 1/25/12 11:36 AM

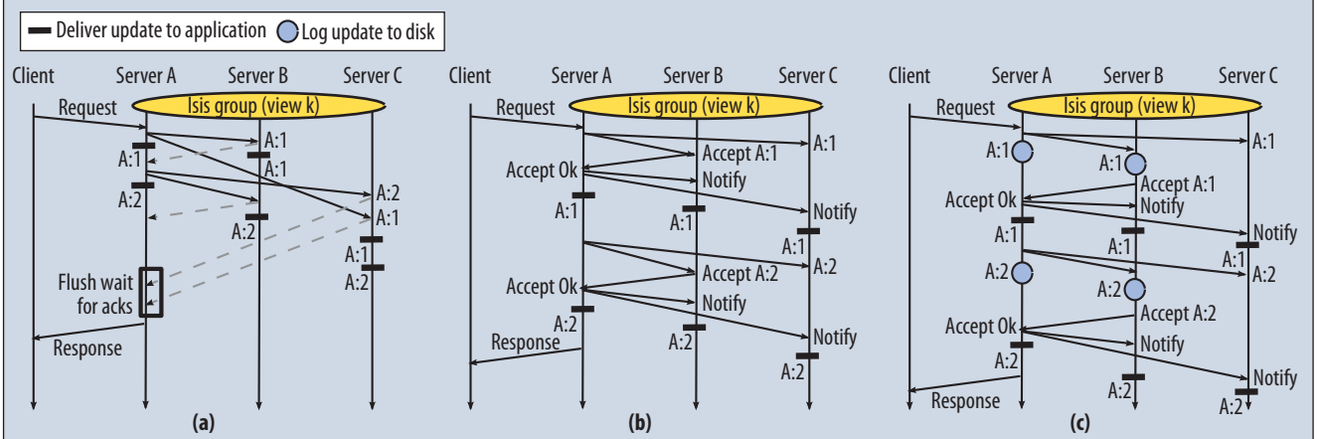


Figure 1. Communication between three members of an Isis² process group for various primitives: (a) Send, followed by a Flush barrier, (b) SafeSend (in-memory Paxos), and (c) durable (disk-logged) Paxos. A:1 and A:2 are two updates sent from server A.

The degree of durability will depend on how this delaying step is implemented. A service unwilling to tolerate any delay might respond while updates are still queued in the communication layer. Its responses to the client would reflect the orderly application of updates, but a crash could cause some updates to be forgotten. With amnesia freedom, other replicas back up an update: only the crash of the entire shard or service could result in their loss. Given that the cloud computing community has ruled out simply waiting for a durable inner-tier service to acknowledge the updates, this offers an appealing compromise: a surprisingly scalable, inexpensive, and resilient new option for first-tier services that need consistency.

THE ISIS² SYSTEM

Our Isis² system supports virtually synchronous process groups,[6] and includes reliable multicasts with various ordering options (available at www.cs.cornell.edu.ken/isis2). The Send primitive is FIFO-ordered. An Ordered-Send primitive guarantees total order; we will not be using it here because we assume that sharded data has a primary copy. Amnesia freedom is achieved by invoking a barrier primitive called Flush that delays until any prior unstable multicasts have reached all destinations. This kind of Flush sends no extra messages; it just waits for acknowledgments (Figure 1a).

Isis² also offers a virtually synchronous version of Paxos,[7] via a primitive we call SafeSend. The user can specify the size of the acceptor set; we favor the use of three acceptors, but it is possible to select all members in a process group to serve as acceptors, or half of those members, or whatever is most appropriate to a given application. SafeSend offers two forms of durability: *in-memory* durability, which we use for soft-state replication in the first tier, and true *on-disk* durability. Here, we evaluate only the in-memory configuration, since the stronger form of durability is not useful in the first tier.

Figure 1 illustrates these protocol options. Figure 1a shows an application that issues a series of Send operations and then invokes Flush, which causes a delay until all the prior Sends have been acknowledged. In this particular run, updates A:1 and A:2 arrive out of FIFO order at member C, which delays A:2 until A:1 has been received; we illustrate this case to emphasize that implementing FIFO ordering is very inexpensive. Figure 1b shows our in-memory Paxos protocol with two acceptors (nodes A and B) and one additional member (C); all three are learners. Figure 1c shows this same case, but with the durable version of the Paxos protocol, emphasizing that in a durable mode, the protocol must store pending requests on disk rather than in memory.

We ran our experiment in a datacenter with 48 machines, each with 12 Xeon X5650 cores (running at 2.67 GHz) and connected by Gigabit Ethernet. We designed a client to trigger bursts of work during which five multicasts are initiated—for example, to update five replicated data objects. Two cases are considered: for one, the multicasts use Send followed by a Flush; the other uses five calls to SafeSend, in its in-memory configuration with no Flush.

Figure 2a shows the latency between when processing started at the leader and when update delivery occurred, on a per-update basis. All three of these consistent-replication options scale far better than might be expected on the basis of the reports in the literature, but the amnesia-free Send significantly outperforms SafeSend even when configured with just three acceptors. The delay associated with Flush was dwarfed by other sources of latency variance, and we did not measure it separately. Notice that if we were sharding data by replicating it to just three to five members, all of these options would offer good performance.

r2bir.indd 55 1/25/12 11:36 AM

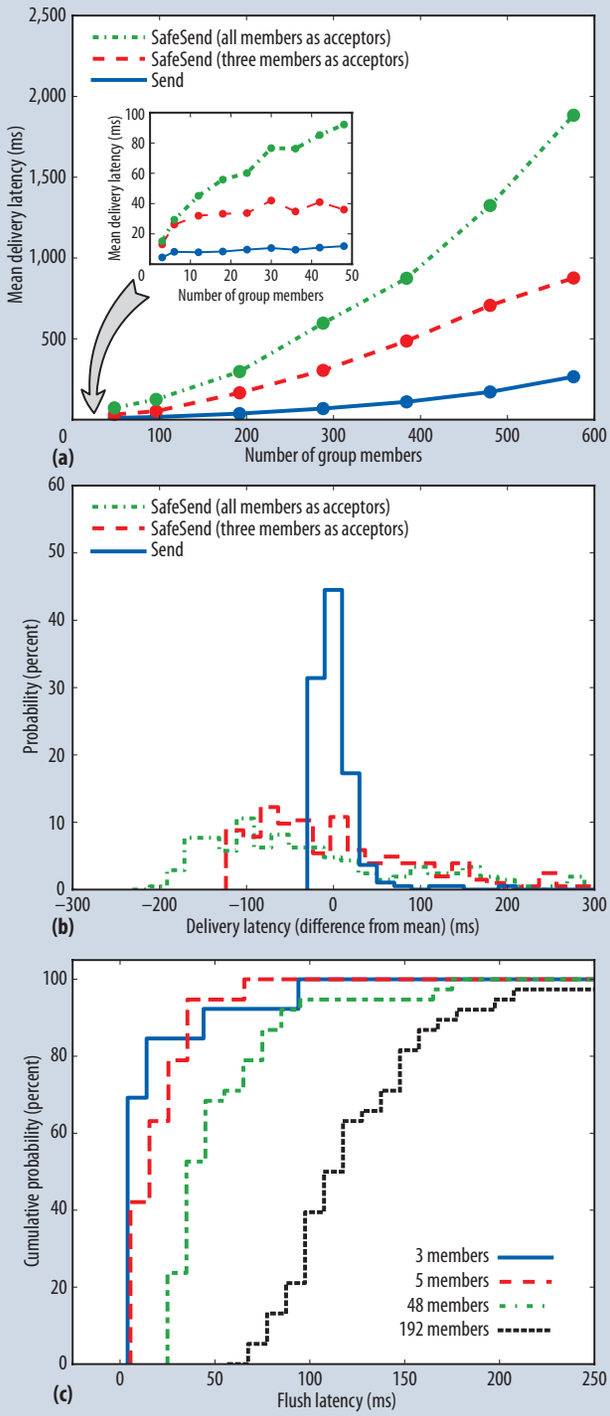


Figure 2. Latency between the start of processing and update delivery. (a) Mean delivery latency for the Send primitive, contrasted with SafeSend with three acceptors and SafeSend with acceptors equal to the number of members in the group. (b) Histogram showing the probability of delivery latencies relative to mean update latency, for all three protocols, for a group size of 192 members. (c) Cumulative histogram of Flush latency for various numbers of members in the group.

In Figure 2a we omitted error bars, and instead plotted variance from the mean in Figure 2b. This figure focuses on the case of 192 members; for each protocol we took the mean as reported on Figure 2a and then binned individual latencies for the 192 receivers as deltas from the mean, which can thus be positive or negative. While Send latencies are sharply peaked around the mean, the protocol does have a tail extending to as much as 100 ms but impacting only a small percentage of multicasts.

For SafeSend the latency deviation is both larger and more common. These observations reflect packet loss: in loss-prone environments—for example, cloud-computing networks, which are notoriously prone to overload and packet loss—each protocol stage can drop messages, which must then be retransmitted. The number of stages explains the degree of spread: SafeSend latencies spread broadly, reflecting instances that incur zero, one, two, or even three packet drops (Figure 1b). In contrast, Send has just a single phase (Figure 1a), hence is at risk of at most loss-driven delay. Moreover, Send generates less traffic, resulting in a lower loss rate at the receivers. In the future, we plan to report on SafeSend performance with disk logging.

ANALYSES OF CAP TRADEOFFS

In addition to Eric Brewer's original work[1] and reports by others who advocate for BASE,[3,4] the relevant analyses of CAP and the possible tradeoffs (CA/CP/AP) include Jim Gray's classic analysis of ACID scalability,[12] a study by Hiroshi Wada and colleagues of NoSQL consistency options and costs, and other discussions of this topic.[13-17] Database research that relaxes consistency to improve scalability includes the Escrow transaction model,[18] PNUTS,[19] and Sagas.[20]

At the other end of the spectrum, notable cloud services that scale well and yet offer strong consistency include the Google File System,[21] Bigtable,[22] and Zookeeper.[23] Research focused on Paxos performance includes the Ring-Paxos protocol[24,25] and the Gaios storage system.[26]

Our work employs a model that unifies Paxos (state-machine replication) with virtual synchrony.[7] In addition to our prior work on amnesia freedom,[6] other mechanisms that we have exploited in Isis[2] include the IPMC allocation scheme from Dr. Multicast,[27] and the tree-structured acknowledgments used in QuickSilver Scalable Multicast.[2,28]

The CAP theorem centers on concerns that the ACID database model and the standard durable form of Paxos introduce unavoidable delays. We have suggested that these delays are actually associated with durability, which is not a meaningful goal in the cloud's first tier, where applications are limited to soft state. Nonetheless, an in-memory form of durability is feasible. Leveraging this, we can offer a spectrum of consistency

r2bir.indd 56 1/25/12 11:36 AM

options, ranging from none to amnesia freedom to strong *f*-durability (an update will not be lost unless more than *f* failures occur). It is possible to offer ordered-based consistency (state machine replication), and yet achieve high levels of scalable performance and fault tolerance.

Although the term amnesia freedom is new, our basic point is made in many comparisons of virtual synchrony with Paxos. A concern is that cloud developers, unaware that scalable consistency is feasible, might weaken consistency in applications that actually need strong guarantees.

Obviously, not all applications need the strongest forms of consistency, and perhaps this is the real insight. Today's cloud systems are inconsistent by design because this design point has been relatively easy to implement, scales easily, and works well for the applications that earn the most revenue in today's cloud. The kinds of applications that need stronger assurance properties simply have not yet wielded enough market power to shift the balance. The good news, however, is that if cloud vendors ever tackle high-assurance cloud computing, CAP will not represent a fundamental barrier to progress. C

Acknowledgments

We are grateful to Robbert van Renesse, Dahlia Malkhi, the students in Cornell's CS7412 class (spring 2011), and to the DARPA MRC program, which funds our efforts.

References

1. E. Brewer, "Towards Robust Distributed Systems," *Proc. 19th Ann. ACM Symp. Principles of Distributed Computing*. (PODC 00), ACM, 2000, pp. 7-10.
2. S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *ACM SIGACT News*, June 2002, pp. 51-59.
3. D. Pritchett, "BASE: An Acid Alternative," *ACM Queue*, May/June 2008, pp. 48-55.
4. W. Vogels, "Eventually Consistent," *ACM Queue*, Oct. 2008, pp. 14-19.
5. G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-Value Store," *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles* (SOSP 07), ACM, 2007, pp. 205-220.
6. K. Birman, "History of the Virtual Synchrony Replication Model," *Replication: Theory and Practice*, LNCS 5959, Springer, 2010, pp. 91-120.
7. K.P. Birman, D. Malkhi, and R. van Renesse, *Virtually Synchronous Methodology for Dynamic Service Replication*, tech. report MSR-2010-151, Microsoft Research, 2010.
8. K. Birman and T. Joseph, "Exploiting Virtual Synchrony in Distributed Systems," *Proc. 11th ACM Symp. Operating Systems Principles* (SOSP 87), ACM, 1987, pp. 123-138.
9. L. Lamport, "Paxos Made Simple," *ACM SIGACT News*, Dec. 2008, pp. 51-58.
10. L. Lamport, "The Part-Time Parliament," *ACM Trans. Computer Systems*, May 1998, pp. 133-169.
11. R. van Renesse, F.B. Schneider, "Chain Replication for Supporting High Throughput and Availability," *Proc. 6th Symp. Operating Systems Design & Implementation* (OSDI 04), Usenix, 2004, pp. 7-7.
12. J. Gray et al., "The Dangers of Replication and a Solution," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD 96), ACM, 1996, pp. 173-182.
13. H. Wada et al., "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: The Consumers' Perspective," *Proc. 5th Biennial Conf. Innovative Data Systems Research* (CIDR 11), ACM, 2011, pp. 134-143.
14. D. Kossman, "What Is New in the Cloud?" keynote presentation, *6th European Conf. Computer Systems* (EuroSys 11), 2011; http://eurosys2011.cs.uni-salzburg.at/pdf/eurosys11-invited.pdf.
15. T. Kraska et al., "Consistency Rationing in the Cloud: Pay Only When It Matters," *Proc. VLDB Endowment* (VLDB 09), ACM, 2009, pp. 253-264.
16. M. Brantner et al., "Building a Database on S3," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD 08), ACM, 2008, pp. 251-264.
17. D. Abadi, "Problems with CAP, and Yahoo's Little Known NoSQL System," blog; http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html.
18. P.E. O'Neil, "The Escrow Transactional Method," *ACM Trans. Database Systems*, Dec. 1986, pp. 405-430.
19. B.F. Cooper et al., "PNUTS: Yahoo!'s Hosted Data Serving Platform," *Proc. VLDB Endowment* (VLDB 08), ACM, 2008, pp. 1277-1288.
20. H. Garcia-Molina and K. Salem, "Sagas," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD 87), ACM, 1987, pp. 249-259.
21. S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *Proc. 19th ACM Symp. Operating Systems Principles* (SOSP 03), ACM, 2003, pp. 29-43.
22. F. Chang et al., "Bigtable: A Distributed Storage System for Structured Data," *Proc. 7th Usenix Symp. Operating Systems Design and Implementation* (OSDI 06), Usenix, 2006, pp. 205-218.
23. F.P. Junqueira and B.C. Reed, "The Life and Times of a Zookeeper," *Proc. 21st Ann. Symp. Parallelism in Algorithms and Architectures* (SPAA 09), ACM, 2009, pp. 4-4.
24. P. Marandi, M. Primi, and F. Pedone, "High-Performance State-Machine Replication," *Proc. IEEE/IFIP Int'l Conf. Dependable Systems and Networks* (DSN 11), IEEE CS, 2011, pp. 454-465.
25. P.J. Marandi et al., "Ring-Paxos: A High-Throughput Atomic Broadcast Protocol," *Proc. IEEE/IFIP Int'l Conf. Dependable Systems and Networks* (DSN 10), IEEE CS, 2010, pp. 527-536.
26. W.J. Bolosky et al., "Paxos Replicated State Machines as the Basis of a High-Performance Data Store," *Proc. 8th Usenix Symp. Networked Systems Design and Implementation* (NSDI 11), Usenix, 2011, pp. 141-154.
27. Y. Vigfusson et al., "Dr. Multicast: Rx for Data Center Communication Scalability," *Proc. 5th European Conf. Computer Systems* (EuroSys 10), ACM, 2010, pp. 349-362.
28. K. Ostrowski, K. Birman, and D. Dolev, "QuickSilver Scalable Multicast (QSM)," *Proc. 7th IEEE Ann. Int'l Symp. Network Computing and Applications* (NCA 08), IEEE, 2008, pp. 9-18.

Kenneth P. Birman *is the N. Rama Rao Professor of Computer Science at Cornell University. His research focuses on high assurance for cloud computing and other scalable*

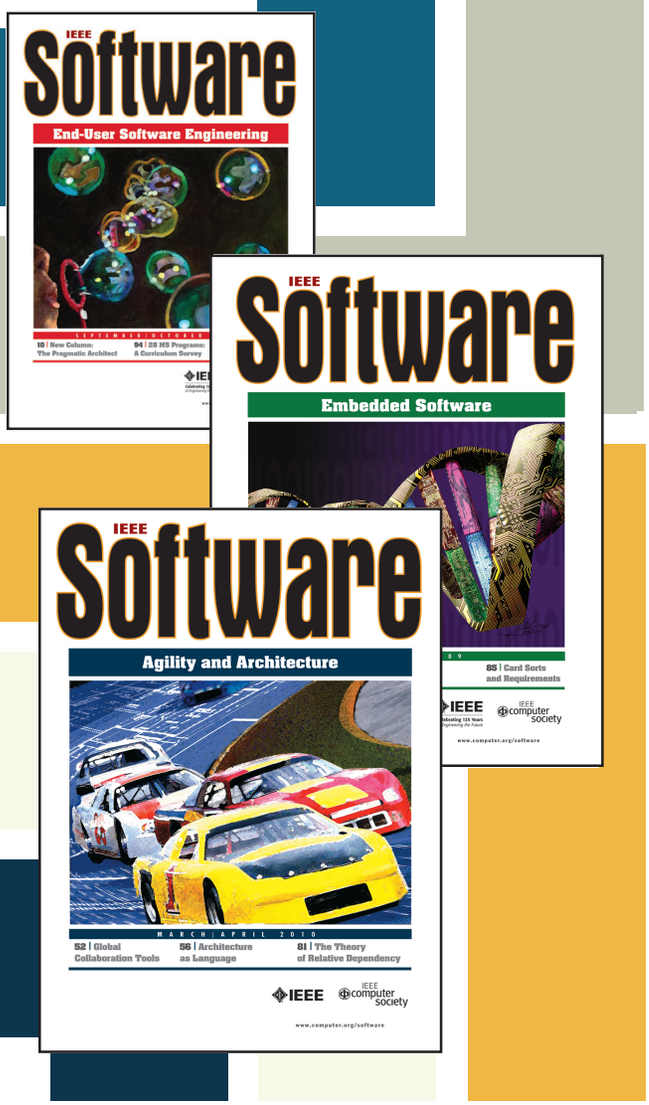
r2bir.indd 57 1/25/12 11:36 AM

distributed systems. Birman received a PhD in computer science from University of California, Berkeley. He received the 2009 IEEE Tsukomo Kanai award for his work in distributed systems and is an ACM Fellow. Contact him at ken@cs.cornell.edu.

***Daniel A. Freedman** is a postdoctoral research associate in the Department of Computer Science at Cornell University. His research interests span distributed systems and communication networks. Freedman received a PhD in theoretical physics from Cornell University. He is a member of IEEE, ACM SIGOPS and SIGCOMM, and Usenix. Contact him at dfreedman@cs.cornell.edu.*

***Qi Huang** is a computer science PhD student at Cornell University. His research interests include distributed computing, networking, and operating systems. Huang received a BE in computer science from Huazhong University of Science and Technology, China. He is a student member of ACM. Contact him at qhuang@cs.cornell.edu.*

***Patrick Dowell** is a computer science ME student at Cornell University. His research interests include distributed systems, cloud computing, and security. Dowell received a BS in computer science from Cornell University. Contact him at pkd3@cs.cornell.edu.*



SUBSCRIBE TODAY

IEEE SOFTWARE

offers pioneering ideas, expert analyses, and thoughtful insights for software professionals who need to keep up with rapid technology change. It's the authority on translating software theory into practice.

www.computer.org/software/SUBSCRIBE

r2bir.indd 58 1/25/12 11:36 AM