Overcoming CAP with Consistent Soft-State Replication

Article *in* Computer · February 2012

DOI: 10.1109/MC.2011.387 · Source: DBLP

Qi Huang
Keele University
14 PUBLICATIONS **153** CITATIONS

Overcoming CAP with Consistent Soft-State Replication

Kenneth P. Birman, Daniel A. Freedman, Qi Huang and Patrick Dowell[1]

1. Abstract

CAP explores tradeoffs between {Consistency, Availability and Partition tolerance}, concluding that a replicated service can possess just two of the three. The theorem is proved by forcing a replicated service to respond to conflicting requests during a partitioning failure, triggering inconsistency. But there are replicated services for which the applicability of CAP is unclear. Here, we look at scalable "soft-state" services that run in the *first-tier* of a single cloud-computing data center. The puzzle is that such services live in a single data center and run on redundant networks. Partitioning events involve single machines or small groups and are treated as node failures; thus, the CAP proof doesn't apply. Nonetheless, developers believe in a generalized CAP "folk theorem," holding that scalability and elasticity are incompatible with strong forms of consistency. We present a first-tier consistency alternative that replicates data, combines agreement on update ordering with *amnesia freedom,* and supports both good scalability and fast response.

2. Introduction

The CAP theorem [8][15] has been influential within the cloud computing community, and motivates the creation of cloud services with weak consistency properties. While the theorem focuses on services that span partition-prone network links, CAP is also cited in connection with BASE (Basically Available replicated Soft state with Eventual consistency), a methodology in which services that run in a *single* data center on a reliable network are engineered to use a non-transactional coding style, tolerate potentially stale or incorrect data, and eschew synchronization. eBay invented the approach [26], and Amazon points to the self-repair mechanisms in the Dynamo key-value store as an example of how eventual consistency behaves in practice [29].

The "first-tier" of the cloud, where this issue is prominent, is in many ways an unusual environment. When an incoming client's request is received, *fast response* is the overwhelming priority, even to the extent that other properties might need to be weakened. Cloud systems host all sorts of subsystems with strong consistency guarantees, including databases and scalable global file systems, but they reside "deeper" in the cloud, shielded from the heaviest loads by the first-tier.

To promote faster responsiveness, first-tier applications often implement replicated in-memory key-value stores, using them to store state or to cache data from services deeper in the cloud. When this data is accessed while processing a client request, locking is avoided, as are requests to inner services, for example to check that cached data isn't stale. To the extent that requests have side-effects that require updates to the cloud state, these are handled in a staged manner: the service member performs the update locally and responds to the client. Meanwhile, in the background (asynchronously), updates are propagated to other replicas and, if needed, to inner-tier services that hold definitive state. Any errors are detected and cleaned up later, hopefully in ways external clients won't notice: *eventual consistency*.

This works well for eBay, Facebook and other major cloud providers today. But will it work for tomorrow's cloud applications? For example, as applications such as medical records management (including "active" medical applications that provide outpatient monitoring and control), transportation systems (smart highways), control of the emerging smart power grid, and similar tasks shift to the cloud, we'll confront a wave of first-tier applications that may be required to justify their actions. Is a scalable solution to this problem feasible?

Our approach mimics many aspects of today's BASE solutions, but offers first-tier service replicas a way to maintain consistent replicated data, across the copies. Locks are not required: any service instance sees a consistent snapshot of the first-tier state, using locks only if a mutual exclusion property is required. To maximize performance, updates are performed optimistically and asynchronously, but to avoid the risk that updates might be lost in a later crash, we introduce a *flush barrier*: before replying to an external client (even for a read-only operation), the primary checks to make sure that any unstable updates on which the request depends have reached all the relevant replicas. Jointly, these techniques yield a strong consistency guarantee and *amnesia freedom:* in-memory durability. We believe this is a good match to the performance, scalability and responsiveness needs of first-tier cloud services.

This paper won't have room to consider other alternatives, such as full-fledged transactions. Nonetheless, our experimental work supports the view that full-fledged atomic multicast wouldn't scale well enough for use in this setting (i.e. durable versions of Paxos, or ACID transactions). As we'll show, strongly durable multicast exhbits marked performance degradation in larger-scale settings. In contrast, amnesia freedom scales well, overcoming the limitations of CAP.

3. Life in the First-tier

Cloud-computing systems are generally structured into tiers: a first-tier that handles incoming client requests (from browsers, applications using web-services standards, etc), caches and key-value stores that run near the first-tier, and inner-tier services that provide database and file-system functionality. A wide variety of back-end applications run off the critical path, preparing indices and other data for later use by online services.

[1] Dept. of Computer Science; Cornell University, Ithaca NY 14850. Emails: {ken,dfreedman,qhuang,pdk3}@cs.cornell.edu. +1-607-255-9199

In the introduction, we discussed some aspects of the first-tier programming model: aggressive replication, very loose coupling between replicas, optimistic local computation without locking, and asynchronous updates. Modern cloud-development platforms standardize this model, and in fact take it even further. In support of elasticity, first-tier applications are also required to be *stateless*: cloud platforms launch each new instance in a standard initial state, and they discard local data when an instance fails or is halted. These terms require some explanation. “Stateless” doesn’t mean that these instances have no local data but rather that they are limited to non-durable *soft state*. On launch, a new instance initializes itself by copying data from some operational instance, or by querying services residing deeper in the cloud. Further, “elasticity” doesn’t mean that a service might be completely shut down without warning: cloud-management platforms can keep some minimum number of replicas of each service running (ensuring continuous availability). Subject to these constraints, however, replication degree can vary rapidly.

These observations enable a form of durability. Data replicated within the soft state of a service, in members that the management platform won’t shut down (because they reside within the core replica set), will remain available unless a serious failure causes all the replicas to crash simultaneously. Failures of that kind will be rare in a well-designed application; we’ll leverage this observation below.

This, then, is the context within which CAP is often cited as a generalized principle. In support of extremely rapid first-tier responses and fault-tolerance, developers have opted for relaxed consistency. Our paper will show that one can achieve similar responsiveness while delivering stronger consistency guarantees by adopting a consistency model better matched to the characteristics of the first-tier. The approach could enable cloud-hosting of applications that need strong justification for any responses they provide to users. 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 won’t make.

4. Consistency: A Multi-Dimensional Property

Terms like *consistency* can be defined in many ways. In prior work on CAP, the “C” is defined by citing the database ACID model (ACID stands for Atomicity, Consistency, Isolation and Durability; 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 papers also point to Paxos [21], an atomic multicast protocol that provides total ordering and durability.

Durability is the guarantee that if an update has been performed it will never be lost. Normally, the property is expected to apply even if an entire service crashes and then restarts. But notice that for a first-tier service, durability in this strongest sense conflicts with the soft-state limitation. By focusing on techniques that guarantee durability and are often used in hard-state settings, one risks reaching conclusions that relate to features not needed by first-tier services. With this in mind let’s review other common but debatable assumptions:

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 live 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 notice that because cloud platforms vary this set elastically, the actual collection will change over time, perhaps rapidly. Full replication forces us to track the set, to have a policy for initializing a newly launched service instance, and to ensure that each update reaches all the replicas, even if that set is 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 whereby any instance can issue read or update requests to any shard. Moreover, since shard membership can change, we’ll need to factor membership dynamics into the model.

One way to handle such issues is seen in Amazon’s Dynamo key-value store [11], 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).* Coordination with the cloud-management service minimizes abrupt elasticity decisions that would shut down shards without first letting members transfer state to new owners.

A second way to implement shards arises in systems that work with *process groups* [4]: here, the various requirements are solved by a group communication infrastructure (such as new Isis^2 system). Systems of this sort offer an API with basic 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 built-in synchronization (Isis^2, for example, implements the virtual synchrony model [5]). The developer decides how shards should work, then uses the provided API to implement the desired policy. With group communication systems, group membership change is a potentially costly event, but a single membership update can potentially cover many changes. Accordingly, use of this approach presumes some coordination with the cloud management infrastructure so that changes are done in batches. Assuming that the service isn’t buggy, the remaining rate of failures should be very low.

Yet a third shard-implementation option is seen in services that run on a stable set of nodes for a long period, enabling a kind of *static* membership in which some set of nodes is designated as running the service. Here, membership remains fixed, but some nodes may 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 isn’t needed unless the static membership is reconfigured.

Several CAP papers express concern about the high cost of quorum operations, especially if they occur on the critical path for end-user request processing. But notice that quorum operations are needed only in the static membership case (not for DHT or process group approaches). Those avoid the need for quorums because they evolve shard membership as nodes join, leave or fail. This avoids a kind of non-local interaction that can be as costly as locking: if every read or update operation on the critical path entails interaction with multiple nodes, the reply to the end-user could be delayed by a substantial number of multi-node protocol events (because many requests will perform multiple reads and updates). With dynamic membership, we gain the ability to do reads and writes *locally* at the price of more frequent group membership updates.

Update ordering. A second dimension of consistency concerns the policy whereby updates are applied to replicas. A *consistent* replication scheme is one that applies the same updates to every replica in the same order, and that specifies the correct way to initialize new members, or nodes recovering from a failure [3][4][5].

Update ordering costs depend on the pattern whereby updates are issued. In many systems, each data item has a primary copy through which updates are routed. Some systems shift the role of being primary around, but the basic idea is the same: in both cases, by delivering updates in the order they occur at the primary, without gaps, the system can be kept consistent across multiple replicas. The required multicast ordering mechanism is simple and very inexpensive.

A more costly multicast-ordering need arises 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, at which point the delivery order can be used to apply the updates in a consistent order at each of the replicas. This is relevant only because the CAP and BASE point to protocols that do permit concurrent updates. Thus by requiring replicated data to have a primary copy, we can achieve 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 won't forget it. But precisely what does it mean to have "performed" an update? And must the durability mechanism retain data across complete shutdowns of the full membership of a service or shard?

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 would push updates to their respective logs, then acknowledge that they are ready to commit the update, and then in a second phase, the updates in the logs can be applied to the actual database. Lamport's Paxos protocol [20][21] doesn't talk about the application per-se, but most implementations of Paxos incorporate this sort of logging of pending updates. Call this *strong* durability, and notice that it presumes durable storage that will survive failures.

But recall that first-tier services are required to be stateless. Can a first-tier service replicate data in a way that offers a meaningful durability property? The obvious possibility is 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. If we call the former solution *non-durable* (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), what should we call this other solution? This will be the case we're referring to as *amnesia freedom:* the service won't forget the update unless all *n* members fail (as noted earlier, that will be rare). Notice that with amnesia freedom, any subsequent requests issued by the client, after seeing a response to a first request, will be handled by service instances "aware" of the updates triggered by that first request.

Amnesia freedom isn't perfect. If a serious failure does force an entire service or shard to shut down, unless the associated data is backed up on some inner-tier service, state will be lost. But, if such events are rare, the risk may be acceptable. For example, suppose that applications for monitoring and controlling embedded systems such as medical monitoring devices move to cloud-hosted settings. While these roles do require consistency and other assurance properties, the role of monitoring is a continuous online one. Moreover, applications of this sort generally revert to a fail-safe mode when active control is lost. Here, one might not need any inner-tier service at all.

Applications that push updates to an inner service have a choice: they can wait for the update to be acknowledged, or could adopt amnesia freedom, but in so doing, accept a window of vulnerability for the period between when the update is fully replicated in the memory of the first-tier service, until it reaches the inner-tier. Another database analogy comes to mind: database mirroring is often done by asynchronously streaming a log, despite the small risk that a failure could cause updates to be lost. An amnesia-free approach has an analogously small risk: having replicated an update in the memory of a first-tier service, the odds of it being lost will be orders of magnitude smaller than in today's BASE approaches.

Failure model. Our work assumes that applications fail by crashing, and that network packets can be lost, and that partitioning failures that isolate a node, or even a rack or container are mapped to crash failures: when the network is repaired, the nodes that had been isolated will be forced to restart. We also assume that while isolated by a network outage, nodes are unable to communicate with external clients.

Putting it all together. We arrive at a rather complex set of choices and options, from which one can construct a diversity of replication solutions with very different properties, required structure, and expected performance. Some make little sense in the first-tier; others represent reasonable options:

- One can 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 which might apply them in different orders.
- One can 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). Amnesia freedom is achieved by enforcing a "barrier": prior to sending a reply to the client request, the system

pauses, 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. If all updates are already stable, no delay is incurred.

- One can implement a strongly-durable state-machine replication model. Most implementations of Paxos use this model. In our target scenario, of course, the strongest forms of durability just aren't meaningful: there is no benefit to logging messages other than in the memory of the replicas themselves.
- One can implement database transactions in the first-tier, coupling them to the serialization order used by inner-tiers, for example via a true multi-copy model based on the ACID model or a *snapshot-isolation* model. This approach is receiving quite a bit of attention in the research community.

How does CAP deal with this diversity of options? The question is easier to pose than to answer. Brewer's 2000 PODC keynote [8] proposed CAP as a general principle. His references to consistency evoke ACID database properties. Gilbert and Lynch offered their proof [15] in settings with partitionable wide-area links, and in fact pointed out that with even slight changes, CAP ceases to apply (for example, they propose a model called t-eventual consistency that avoids the CAP tradeoff). In their work on BASE, Pritchett [26] and Vogels [29] point both to the ACID model and to the durable form of Paxos [20][21]. They argue that these models will be too slow for the first-tier; their concerns apparently stem from the costly two-phase structure of these particular protocols, and from their use of quorum reads and updates, resulting in delays on the critical path that computes responses.

Notice that the performance and scalability concerns in question stem from durability mechanisms, not those supporting order-based consistency. As we saw earlier, sharded data predominates in the first-tier, and one can easily designate a primary copy at which updates are performed first. Other replicas mirror the primary. Thus the "cost of consistency" can be reduced to the trivial requirement that updates be performed in the same FIFO order used by the primary.

This then leads to our proposal. We recommend that first-tier services employ a shared model, with a primary replica for each data item, and we favor a process-group model that coordinates group membership changes with updates: virtual synchrony [4]. When a first-tier service instance receives a request, it executes it using local data, and applies any updates locally as well, issuing a stream of asynchronous updates that will be delivered and applied in FIFO order by other replicas. This permits a rapid but optimistic computation of the response to the user: optimistic not because any rollback might be needed, but because failure could erase the resulting state. This risk is eliminated by imposing a synchronization barrier prior to responding to the client, even for read-only requests. The barrier delays the response until any prior updates have become stable, yielding a model with strong consistency and amnesia freedom. Relative to today's first-tier model, the only delay is that associated with the barrier event.

5. The Isis^2 System

In this section, we offer a brief experimental evaluation of the system, focused on the costs of our proposed scheme. Our new Isis^2 system supports virtually synchronous process groups and includes reliable multicasts with various ordering options. The **Send** primitive is per-sender FIFO ordered. An **OrderedSend** primitive guarantees total order; we won't be using it here because we're assuming that sharded data has a primary copy. The barrier primitive is called **Flush;** it waits passively until prior multicasts become stable.

Isis^2 also supports a virtually synchronous version of Paxos [5], via a primitive we call **SafeSend**. The user can specify the size of the acceptor set; we favor the use of three acceptors, but one could certainly select all members in a process group to serve as acceptors, or some other threshold appropriate to the 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 only evaluate the in-memory configuration.

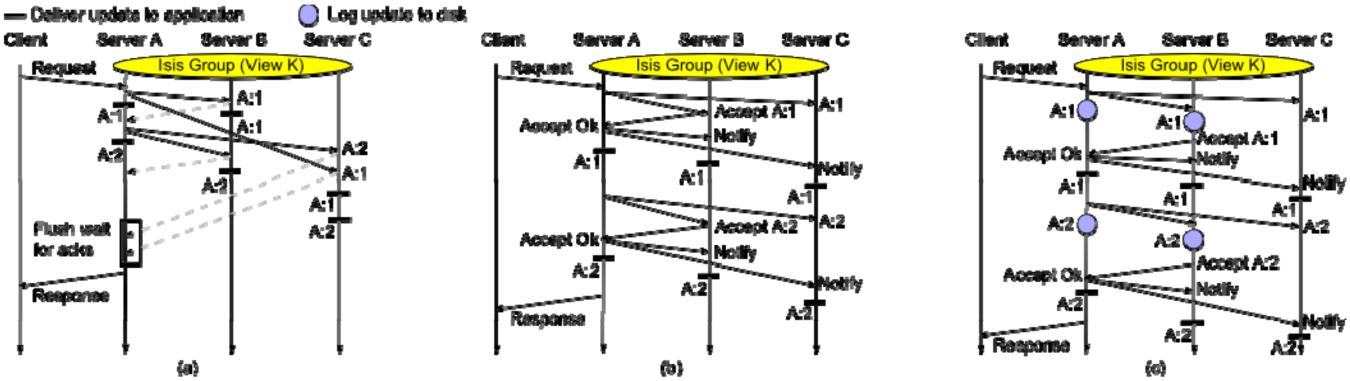

Figure 1: Communication between three members of an Isis2 process group for various primitives: (a) Send, together with a Flush barrier that prevents clients from glimpsing unstable states. (b) SafeSend (in-memory Paxos). (c) Durable (disk-logged) Paxos. A:1 and A:2 are two updates sent from Server A. These runs all use a single-threaded sender; with multiple threads SafeSend would have a more overlapped pattern of traffic, that wouldn't impact the per-invocation latency metrics on which our evaluation focuses. Our key insight is that for soft-state replication, SafeSend is no stronger than that of Send+Flush.

Figure 1 illustrates these protocol options. In Figure 1(a) we see 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 just to emphasize that FIFO ordering is needed, but inexpensive to implement. To avoid clutter, Figure 1(a) omits stability messages that normally piggyback on outgoing multicasts: they inform the replicas that prior multicasts have become stable. Thus, some future update, A:3, might also tell the replicas that A:1 and A:2 are stable and can be garbage-collected.

Figure 1(b) shows our in-memory Paxos protocol with 2 acceptors (nodes A and B) and one additional member (node C); all three are *learners* (all deliver messages to the application layer). Figure 1(c) shows this same case, but now with a durable version of the Paxos protocol: pending requests are now logged to disk rather than being held in memory. Not shown is the logic by which the log files are later garbage collected; it requires additional rounds of agreement, but they can be performed offline.

Notice that had we responded to the client in Figure 1(a) without first invoking the **Flush** barrier, even a single failure could cause amnesia; once **Flush** completes, the client response can safely be sent. **Flush** is needed even for a read-only client-request, since a read might observe data that was recently updated by a multicast that is still unstable. On the other hand, a **Flush** would be free unless an unstable multicast is pending. In fact, our experiments consider the worst-case: a pure-update workload; even so, **Flush** turns out to be cheap. For a read-intensive workload, it would be cost even less, since there would often be no unstable multicasts, hence no work to do.

The experiment we performed focuses on the performance-limiting step in our consistent-replication scheme. In many papers on data replication, update throughput would be the key metric. For our work, it is more important to measure the *latency to perform remote updates*, relative to the responsiveness goals articulated earlier. The reasons are dual. On the sender side (e.g. Server A in Figure 1(a)), all costs are local until the call to **Flush**, and the delay for that call will be determined by the latency of the earlier asynchronous **Send** operations. For **SafeSend**, the latency until delivery to the sender and to the other receivers is even more crucial; the sender cannot do local updates until a sufficient set of acceptors acknowledges the first-phase request. Thus, the single metric that tells us the most about the responsiveness of our solution is latency from when updates are sent, to when they are received and the application is able to perform them.

We decided not to introduce membership churn; if we had done so, membership changes would have brought additional group reconfiguration overheads. Isis^2 batches membership changes, applying many at a time. The protocol normally completes in a few milliseconds, hence even significant elasticity events can be handled efficiently.

We ran our experiment on 48 nodes, each with twelve Xeon X5650 cores and connected by Gigabit Ethernet. Experiments with group sizes up to 48 used one member per node; for larger configurations we ran twelve members per-node. We designed a client to trigger bursts of work during which five multicasts are initiated (e.g. 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 2(a) reports 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 one might have expected on the basis of the literature discussed earlier, but the amnesia-free **Send** significantly outperforms **SafeSend** even when configured with just three acceptors.

Our experiments look at replication on two scales. Sharded data is normally replicated in fairly small groups; we assumed these would have a minimum of 3 and a maximum of 48 members (inset in Figure 2(a)). Then we scaled out more aggressively (placing members on each core of each physical node) to examine groups with as many as 576 members. We limited ourselves to evaluating **SafeSend** in its in-memory configuration (Figure 1(b)), leaving studies of durable-mode performance (Figure 1(c)) for future investigation; this decision reflects our emphasis on soft-state services in the first-tier.

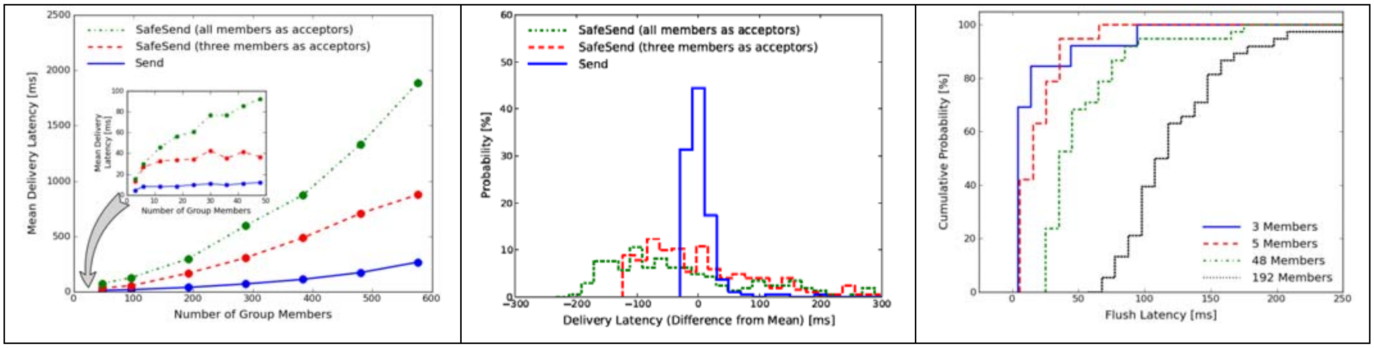

Figure 2(a): Mean delivery latency per single invocation of the Send primitive, contrasted with the same metric for SafeSend with three acceptors and SafeSend with acceptors equal to the size of the group; (b) Histogram of delivery jitter (variance in latency relative to the mean), for all three protocols in a group size of 192 members. (c): CDF of delays associated with the Flush barrier.

For Figure 2(b) we picked one group size, 192 members, and explored variance of delivery latency from its mean, as shown in Figure 2(a). We see that while **Send** latencies are sharply peaked around the mean, the protocol does have a tail extending to as much as 100ms but impacting just 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 (cloud-computing networks 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 (see Figure 1(b)). In contrast, **Send** has just a single phase (Figure 1(a)), and hence is at risk of at most loss-driven delay. Moreover, **Send** generates less traffic, and this results in a lower loss rate at the receivers.

6. Related Work

There has been debate around CAP and the possible tradeoffs (CA/CP/AP). Our treatment focuses on CAP as portrayed by Brewer [8] and by those who advocate for BASE [26][29]. Other relevant analyses include Gray's classic analysis of ACID scalability [16], Wada's study of NoSQL consistency options and costs [30], and Kossman's [18][19][7] and Abadi's [1] discussions of this topic. Database research that relaxes consistency to improve scalability includes the Escrow transaction model [25], PNUTS [12], and Sagas [13]. At the other end of the spectrum, notable cloud services that scale well and yet offer strong consistency include GFS [14], BigTable [10] and Zookeeper [17]. Papers focused on performance of Paxos include the Ring-Paxos protocol [22][23] and the Gaios storage system [6].

Our own work employs a model that unifies Paxos (state-machine replication) with virtual synchrony [5]; in particular, the discussion of amnesia freedom builds on one in [4]. Other mechanisms that we've exploited in Isis^2 include the IPMC allocation scheme from Dr. Multicast [28], and the tree-structured acknowledgements used in QuickSilver Scalable Multicast [24].

7. Conclusions

CAP is centered on concerns that the ACID database model and the standard durable form of Paxos introduce unavoidable delays. Focusing carefully on the consistency and durability needs of first-tier cloud services, we've shown that strong consistency and a form of durability we call amnesia freedom can be achieved with very similar scalability and performance to today's first-tier methodologies. Our approach would also be applicable elsewhere in the cloud.

Obviously, not all applications need the strongest guarantees, and perhaps this is the real insight. Today's cloud systems are inconsistent by design because this design point works well for the applications that earn the revenue in today's cloud. The kinds of applications that need stronger assurance properties simply haven't wielded enough market power to shift the balance. The good news, however, is that if cloud vendors decide to tackle high-assurance cloud computing, CAP will not represent a fundamental barrier to progress.

8. Acknowledgments

The authors are grateful to Robbert van Renesse, Dahlia Malkhi, and to the students in Cornell's CS7412 class (Spring 2011).

9. References

[1] M. Abadi. Problems with CAP, and Yahoo's little known NoSQL system. http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html
[2] A. Adyay, J. Dunagan, A. Wolman. Centrifuge: Integrated Lease Management and Partitioning for Cloud Services. Proc. NSDI 2010.
[3] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. 11th ACM Symposium on Operating Systems Principles, Dec 1987.
[4] K.P. Birman. History of the Virtual Synchrony Replication Model. In *Replication:Theory and Practice.* B. Charron-Bost, F. Pedone, A. Schiper (Eds) Springer Verlag, 2010. Replication, LNCS 5959, pp. 91–120, 2010.
[5] K.P. Birman, D. Malkhi, R. van Renesse. Virtually Synchronous Methodology for Dynamic Service Replication. Submitted for publication. November 18, 2010. Also available as Microsoft Research TechReport MSR-2010-151.
[6] W.J. Bolosky, D. Bradshaw, R.B. Haagens, N.P. Kusters and P. Li. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. NSDI 2011.
[7] M. Brantner, D. Florescu, D. Graf, D. Kossmann and T. Kraska. Building a Database on S3. ACM SIGMOD 2008, 251-264,
[8] E. Brewer. Towards Robust Distributed Systems. Keynote presentation, ACM PODC 2000.
[9] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06). USENIX Association, Berkeley, CA, USA, 335-350.
[10] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06). USENIX Association, Berkeley, CA, USA, 205-218.
[11] G. DeCandia, G., *et. al.* Dynamo: Amazon's highly available key-value store. In Proceedings of the 21st ACM SOSP (Stevenson, Washington, October 2007).
[12] B. Cooper, et. al. PNUTS: Yahoo!'s hosted data serving platform, Proc. VLDB 2008, 1277-1288.
[13] H. Garcia-Molina and K. Salem. SAGAS. SIGMOD 1987, 249-259.
[14] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. 1^{9th} ACM SOSP (Oct. 2003), 29-43.
[15] S. Gilbert, N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, Volume 33 Issue 2. June 2002
[16] J. Gray, P. Helland, D. Shasha. The Dangers of Replication and a Solution. Proc ACM SIGMOD, 1996.
[17] F. Junqueira; B. Reed. The life and times of a ZooKeeper. Proc. ACM Symposium on Parallel Algorithms and Architectures (SPAA), Aug. 2009
[18] D. Kossman. Keynote talk, Eurosys 2011 (April 2011).

[19] T. Kraska, M. Hentschel, G. Alonso and D. Kossmann, Consistency Rationing in the Cloud: Pay only when it matters. VLDB 2009, 253-264.

[20] L. Lamport. The part-time parliament. ACM TOCS, 16:2. May 1998.

[21] L. Lamport. Paxos made Simple. ACM SIGACT News 32:4 (Dec. 2001), 51-58.

[22] P. J. Marandi, M. Primi, N. Schiper F. Pedone. Ring-Paxos: A High Throughput Atomic Broadcast Protocol. 40th ICDSN, 2010.

[23] P. J. Marandi, M. Primi and F. Pedone. High Performance State-Machine Replication, 41st ICDSN, 2011.

[24] K. Ostrowski, K. Birman, D. Dolev. QuickSilver Scalable Multicast (QSM). IEEE NCA. Cambridge, MA. (July 08).

[25] P.E. O'Neil. The Escrow transactional method. ACM Trans. Database Systems 11:4 (Dec. 86), 405-430.

[26] D. Pritchett. BASE: An Acid Alternative. ACM Queue, July 28, 2008. http://queue.acm.org.

[27] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial Computing Surveys, 22:4, Dec. 1990

[28] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, R. Burgess, H. Li, G. Chockler, Y. Tock. Dr. Multicast: Rx for Data Center Communication Scalability. Eurosys, April 2010 (Paris, France). ACM SIGOPS 2010, pp. 349-362.

[29] W. Vogels. Eventually Consistent - Revisited. Dec 2008. http://www.allthingsdistributed.com

[30] H. Wada, A. Fekete, L. Zhao, K.Lee, A. Liu. Data Consistency Properties and the Tradeoffs in Commercial Cloud Storages. CIDR 2011, Asilomar, Jan. ’11