# Overcoming the "D" in CAP: Using Isis$^2$ To Build Locally Responsive Cloud Services

Kenneth P. Birman, Qi Huang, Dan Freedman

Dept. of Computer Science; Cornell University, Ithaca NY 14850

{ken,qhuang,dfreedman}@cs.cornell.edu. +1-607-255-9199

## ABSTRACT

The CAP theorem establishes that a cloud service can only guarantee two of {Consistency, Availability and Partition Tolerance}, motivating developers to reject transactional ACID properties. Instead, they use BASE: a methodology whereby one transforms an application into a faster and more scalable version by running it as a series of asynchronous steps that each use local data replicas (even if potentially stale), eschew locking, and are designed to tolerate unplanned failures and service launches. Along the way, consistency is substantially weakened.

But CAP and BASE may not be the final word. The new Isis$^2$ platform supports consistent, *locally responsive* cloud services. The system is scalable, highly available, and fast. Responses to client requests can be computed using purely local data, hence delays are limited only by local computational costs. Updates propagate asynchronously and map to a single IP multicast; locking is usually avoided by employing primary-copy replication, and otherwise is performed with an inexpensive token-passing scheme. The approach relaxes durability for soft-state updates: this yields an "ACI and mostly D" model. Durability violations are concealed using a form of firewall.

## 1. INTRODUCTION

Cloud computing has been shaped by the CAP theorem [8][17], which holds that a scalable service for the cloud can have just two out of three of consistency, availability and partition tolerance. BASE, the most important cloud development methodology [26][32], offers a methodology for scalable service development. One takes an application (perhaps, an ACID transaction) and starts by mapping data and processing over the nodes using partitioning and data sharding. The next step breaks sequences of operations into pipelines of one-shot tasks linked by message queues. Finally, these tasks are implemented without locks, issuing replicated updates asynchronously and responding to the user without waiting for them to complete.

The last steps abandon several classic consistency properties; if an application needs strong guarantees, it must implement them on the client side. Weak consistency also can entail weak security: security logics justify decisions by deduction on rules and role bindings; if the underlying data or rule base are stale or incorrect, security violations can occur.

Our work starts with a reexamination of the "C" in CAP: we'll show that CAP-style consistency is expensive not because of the guarantee of consistency, per-se, but rather because of an implicit *durability* guarantee: a rarely discussed "D" in CAP. This leads to an insight: for uses that don't actually need durability we can offer strong consistency without paying high scalability or performance penalties. We obtain a form of state machine replication [28]. However, whereas standard state machine replication and ACID solutions perform poorly in cloud-scale deployments, our very similar model will turn out to scale well. The key is to relax durability under conditions *when it is safe to do so.* But this safety issue won't be obvious. Our solution requires several mechanisms to ensure that if a partitioning fault ever triggers the loss of a soft-state update, the impact is contained, then repaired.

Given that it won't turn out to be completely trivial, is such a model really needed? Many cloud providers have argued that the cloud is just not matched to high-assurance applications. We believe that such views are shortsighted. Operators of critical infrastructure applications are as motivated by cost as everyone else. A recent white paper by the CTO of the United States argued for shifting a range of healthcare, finance, transportation, power grid control, and other systems to the cloud with the hope of achieving vast savings [21]. Such a vision demands a scalable consistency story of a kind currently lacking.

The contributions of this paper proceed from our reanalysis of CAP and BASE. We uncover options beyond the ones normally employed by developers using BASE, and this motivates a new consistency model (ACI and mostly D), supported by our Isis$^2$ system. Isis$^2$ targets the same kinds of cloud settings, with the same ambitious scalability and performance goals, as BASE. To guide the Isis$^2$ developer towards a safe solution, we extend BASE by categorizing soft-state into three cases, then show how to apply our methodology in each. Finally, we undertake a side by side comparison of the resulting style of scalable service with one implemented using only the standard Paxos-style of data replication. Our proposed approach offers dramatic speedups, and yet maintains a single all-encompassing order-based consistency model that enables reasoning both about the correctness of the services we build, and also about the properties of high-assurance applications that use them.

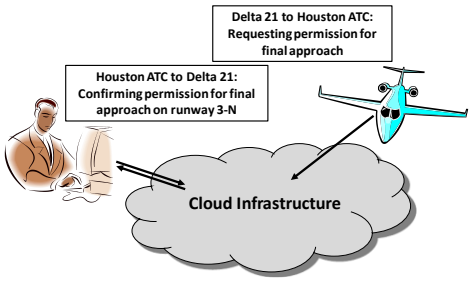


Figure 1: In a high-assurance application, such as an ATC system, every action might have safety implications. CAP and BASE seems to rule out running such systems on the cloud because they weaken consistency. But not all forms of consistency are subject to the CAP theorem. Our work shows that one can adopt a BASE methodology and still achieve a strong form of consistency.

## 2. THE BASE APPROACH

Most readers will be familiar with BASE, an acronym that stands for Basically Available, Soft-State, Eventual Consistency [26][32]. BASE is a methodology; it guides developers to use pipelined transactions, employ weak synchronization and to think about simple fault-tolerance mechanisms, which might not fully mask failures. These steps confer dramatic scalability and performance gains, but they relax consistency.

BASE works for two distinct reasons. One is that for many purposes we can engineer applications to use *soft-state* as a way of reducing the frequency of accessing *hard-state.* As Brewer explains, hard-state corresponds to durable data (i.e. files and databases), while soft-state is derived from other sources. This is an informal definition; in Section 7 we'll be a bit more specific, breaking soft-state into three major subcategories defined by their degree of interdependence on hard-state. Some applications can run almost entirely on soft-state, others may still need hard-state, but can avoid accessing it except when absolutely necessary. The key insight is that *by definition* soft-state has weaker requirements than hard-state, and this makes soft-state highly replicable. For example, we can scale a cache to a massive number of nodes, and yet if a cache record is stale, or is discarded during a failure/restart event for some node, we can always retrieve the correct data from the underlying hard-state backing store.

These benefits come with a catch, because BASE systems often use surprisingly slow soft-state replication techniques. IP multicast (IPMC) is the very fastest way to move high volumes of updates to large numbers of receivers, but most cloud-scale data centers severely restrict IPMC, fearing instabilities [29]. The alternative is to update soft-state using techniques such as TCP chains [30] or gossip protocols of the kind used in Amazon's Dynamo [13] (soft-state replication with eventual consistency). But such approaches are much slower than IPMC, with which an update potentially reaches its targets after just a single network packet latency. Thus BASE solutions often run on state that is somewhat stale.

This creates a risk: in the effort to maximize soft-state and to tolerate staleness, developers can easily arrive at buggy solutions. Acknowledging the problem, BASE encourages developers to consider redefining correct behavior to accommodate responses that might normally be considered incorrect, for example by masking inconsistencies so that the end-user won't notice the issue. This makes sense, because rock-solid guarantees often aren't needed: if a web page looks odd, the user refreshes the page. But one would worry about an air traffic control system that deliberately uses stale data when responding to queries from pilots or controllers.

Today, these concerns suggest that high assurance applications would either be denied the speedups associated with BASE, or if they use a BASE methodology, would need to deploy compensating mechanisms to detect and protect against inconsistencies. Doing so requires application-specific solutions and will often involve complex, ad-hoc, mechanisms. Such steps invite bugs, and could add so much overhead that they end up reintroducing the very same scaling and performance problems that CAP and BASE set out to avoid. Our goal is to offer a platform-level solution that scales as well as BASE but has better performance (by leveraging fast multicast), and does this without allowing buggy, inconsistent behavior.

## 3. CAP IS REALLY ABOUT DURABILITY

CAP is at the core of our work, yet one wonders why the theorem even applies in the cloud. First, the "P" in CAP stands for partition tolerance, yet data centers don't normally need to ride out large-scale partitioning faults. Of course, individual machines (and racks or even containers) do become disconnected from the main system. But when this happens, we don't need to maintain *symmetric* availability. Instead, cloud management systems shut down the isolated machines, leaving the far larger main system running. Later, when the impacted nodes are restarted, they do so in a clean state. Thus "P", in practice, is asymmetric: availability matters only in the main part of the data center.

The CAP theorem employs a symmetric definition for partition tolerance, and that definition plaus a role in the proof. But the BASE papers acknowledge that such faults wouldn't occur in cloud computing data centers [32]. Instead, they suggest a slightly different goal: the creation of services that can maintain availability even if some replicated system components are running with just a subset of the full replica set [26]. There is no need for isolated components to maintain any form of availability: the data center management subsystem will reboot them in any case, once the problem is corrected.

Similarly, there are many ways to define consistency. Brevity precludes a detailed discussion, but these include:

- Atomicity with linearizability, meaning that if operation o' is in the causal future of o, then any replica that executes o' will have first executed o. This is the definition used by Gilbert and Lynch in their proof of the CAP theorem.
- Database consistency and its "cousins": ACID transactions (atomicity, consistency, isolation and durability), or such variants as ε-serializability, state machine replication (Paxos), virtually synchronous process-groups, etc.
- Any of the above, but now qualified by the term "eventual:" if the network is healthy the server guarantees the specified form of consistency, but during network partitioning may respond using a stale state. When the partition ends, consistency is restored within some bounded delay.
- "t-eventual consistency" as defined by Gilbert and Lynch: lineralizability and atomicity. Consistency can be violated during network partitions but is restored within delay t after a partition heals.
- Convergent consistency: Once quiescent, the state reflects all updates (in order) with probability rapidly approaching 1.0 as a function of time.
- Rollback consistency: the system tries to move forward, but might back out a committed operation to correct an inconsistency.
- Best-effort: the system strives to reach a consistent state, but makes no promises.

So, we have a range of consistency options; the CAP theorem applies directly to the first, and would also apply to systems that use standard ACID transactional guarantees. But we there are some forms of consistency for which the theorem simply doesn't apply, and others for which the question is unresolved.

Next, let's think about scalable consistency in a purely practical sense. Today's major cloud computing systems include all sorts of components that offer strong guarantees and play key roles in massively scaled settings. Consider the Google GFS file system

[16] and the systems running on it (BigTable, MapReduce, etc), or Yahoo!'s Zookeeper [18], or other recent systems like Gaios [6]. These offer strong guarantees by adopting some form of hierarchical structure. For example, GFS contains an implementation of Paxos [22] in its Chubby lock service [8]. This is used by the GFS chunk-master, which in turn steers applications to the appropriate chunk-server: a "chain of consistency".

Thus, one can definitely build scalable, strongly consistent services. But none of these were simple to implement. On the contrary, as the teams that built these services emphasize, they needed extensive deployment experience to get their solutions to work in a stable, scaled-out way. The adage is that each new factor of 10 forces a major redesign.

Indeed, CAP is perhaps best understood as a warning to the typical developer (and not addressed to major teams prepared to invest whatever it may take). CAP alerts those "normal" developers that strong consistency is a very costly property to seek in a large-scale application. BASE shows those same developers that for many kinds of services, strong consistency isn't even needed.

Our belief is that CAP and BASE aren't really about consistency or partition tolerance, per se. Rather, we see here an unstated and entirely different goal: that of guiding typical developers towards *scalable, locally responsive, elastic services*. By "locally responsive" we mean that a service replica should be able to act, and respond, without locks or accessing R/W quorums. By "elastic" we mean that it should be possible to dramatically rescale the service with little (or no) warning. If one rereads the CAP and BASE papers with this in mind, it is immediately clear that the authors are discussing local responsiveness. They know how to achieve elastic local responsiveness with strong consistency guarantees, but are also aware of just how hard it can be. Not every application should take the toughest path. Accordingly, they urge the developer to at least consider weakening consistency, because local responsiveness and elasticity are the properties by which the quality of the application will be measured.

But now we arrive at a somewhat surprising insight. It turns out that the crux of the BASE analysis revolves not around consistency per-se, but rather *durability*.

# 4. WHY IS DURABILITY SO EXPENSIVE?

We're using the term local responsiveness for what we see as the key property desired in a scaled-out cloud service: a locally responsive service never needs to wait for anything, hence the end-user is guaranteed the lowest feasible waiting times. Durability, of course, is the property familiar from the ACID model. Durability is also central to atomic multicast protocols such as Paxos, in which a multicast is not delivered until durability can be guaranteed.

A durable update is simply an operation that is guaranteed to be retained: if an update is performed, and later we ask the system to do something, the system will respond from a state that reflects the update. But Brewer's soft/hard-state distinction introduces a new twist, as we'll illustrate with a simple example. Suppose we've just told user A to send future requests to server S. Such requests are often advisory in nature; if the user can't reach S it may be permitted to send requests to some other server. If S fails and then reboots, it might forget the cached state that made it such a good choice for user A's requests. Yet nothing terrible will happen. There are many situations in which the nature of the underlying data makes durability unimportant for at least some kinds of updates. If Amazon slightly bungles a book popularity index, or omits something from a user's browsing history, or even if it puts two copies of a book into the shopping cart, minor annoyance results, but Amazon is still in business. Indeed, in discussing Dynamo, the Amazon design team comments that they made a decision to accept a small risk of such errors to achieve dramatic scalability [32][13]. On the other hand, one would not trust an air traffic control system or a medical care system that sometimes forgets updates.

Our central argument will to be that the kinds of updates for which durability can be lost involve soft-state, whereas one needs durability for to hard-state. The Amazon examples were soft state; the ATC example is talking about hard state. This takes on a special importance when we consider BASE: a methodology centered on soft-state replication, but one in which the assumption is that the only way to make this kind of replication is to adopt a weak consistency model. Our view, in contrast, is that applications benefit from consistency guarantees even for soft-state. Intuition can be drawn from other settings: ask yourself why multi-core systems need coherent caching. Clearly, because many concurrent data structures and algorithms are only correct with coherent caching. In a similar sense, builders of high assurance systems who know that all actions of the platform are consistent are in a better position to reason about the correctness of the application. We'll expand on this in Section 8.

This now leads back to our comment about durability: why should guaranteeing durability pose such a threat to local responsiveness? The CAP and BASE papers both define consistency as a composite property: an order-based guarantee combined with a durability guarantee. Then they point out that durability is costly and at odds with local responsiveness; more specifically, they assert that for durability, we will need to implement read/write quorums. And then they conclude that consistency isn't scalable.

The detailed argument runs this way. Suppose that we replicate data on some set of nodes that will run a cloud computing service, perhaps, $n$ of them. Failures do happen, so now suppose that at some point in time $f$ might be crashed or inaccessible. We'll want our end-user operations to be locally responsive, hence any actions required to perform those operations must advance if just $n-f$ copies are available. Updates will run on $n-f$ replicas, and so reads will need to check $f+1$ replicas. But now we're in trouble: recall that our goal was to guarantee local responsiveness. Accessing multiple replicas, whether to make sure that $n-f$ copies saw an update, or to perform a read on $f+1$ copies, is a costly proposition, particularly when compared with a purely local execution that reads whatever data it finds at the replica handling a request, and fires off its updates purely asynchronously, without waiting before responding to the end-user. Thus, *durability limits the scalability of "atomicity"*.

One can now ask: is durability the *only* such limiting factor? In CAP, consistency also involves locking and request ordering. Won't the need to order conflicting updates also require coordination among sets of replicas? Indeed, this could be an issue, but there are ways to provide ordering without requiring non-local actions. In particular, most large-scale systems favor primary-copy replication scheme in which updates occur first at a primary node, which then relays them to the replicas via a multicast. Thus, in typical cloud settings, a FIFO ordering on updates is all we need. The updates for a given piece of data always come from a single source. *It follows that durability, not the ordering aspects of consistency, was the expensive step.*

Many cloud-computing runtime environments exacerbate this problem by concealing information about virtual node placement: an application running on 5 virtual nodes could easily be scheduled to run on just 1 or 2 physical nodes. Few platforms offer ways to request desired layouts; those that accept such requests often treat them as advisory. The bias runs towards concealing information about placement, and virtual-to-physical node mappings can change without warning. This matters because one could imagine situations in which *n* is very large, and where simply knowing that an update was on 2 or 3 failure-independent nodes would suffice. But in practice, there is often no way to be sure, other than to wait for all *n-f* to acknowledge receipt.

The foregoing analysis is far from comprehensive: it only considers a certain style of state-machine replication. There are many ways to replicate services, and some of them (such as Byzantine Fault Tolerance protocols) have stronger goals than we do here. But we've paralleled the CAP and BASE analysis in order to build the case that one can accept the premises of that work and yet reach a different conclusion.

## 5. "ACI" AND MOSTLY, "D"

Brevity now forces a choice. Within the length limits of this paper we could offer our detailed model, which is a variation of virtual synchrony in which the virtual synchrony process-group model [1][4] is fused with the Paxos-style of state machine replication model. However, we've described this new model elsewhere as part of a joint effort undertaken with Malkhi and Van Renesse [5]. Repeating the formal treatment here would consume most of the space remaining in this paper. Moreover, our goals here are rather practical, focused on building up to the experiments in Section 9, where we put durable and weakly durable updates side by side and compare their relative scalability and latency.

So we'll just summarize the salient aspects of the model. The model starts by pinning down the assumptions we make about failures (crashes), the precise definition of consistency we use (a form of linearizability linked to a weaker notion of atomicity than the one employed in CAP and BASE), and the precise meaning of durability in this context (durable updates are those that can't be lost after a failure). Then the model shows that a state-machine style of replication protocol can exist side by side with virtual synchrony view protocols. Doing so offers some advantages over both the earlier work that used Paxos to implement state machine replication, and the prior work with virtual synchrony.

The key step in our model is concerned with membership change, which requires a somewhat elaborate consensus decision. This achieves a strong form of agreement both on the next view, and on messages in the old view, and is performed in such a way that any message received by all members of the old view is included in the state used by the new view (is durable). Any messages not included will either not have been delivered at all or, if they did get to some members, only reached ones that promptly failed.

For our purposes here, the most interesting feature of this merged model is that it eliminates the need for quorums. In our new model, execution of a service runs as a series of epochs. Each epoch is defined by the set of members (replicas). The epoch starts in a well defined state (namely the final state of the prior epoch) and ends at a well-defined point (namely, when the next epoch starts). If members fail, an epoch becomes *wedged* and this triggers a request for membership reconfiguration. *The update waits*. We stress this because it represents a point of departure relative to the kind of system that BASE presumes: in those systems, such an update *proceeds, if it reaches n-f replicas.*

How does the service become unwedged? Our membership protocol runs. The reconfiguration request just mentioned activates our membership service, which interacts with group members. It wedges the remaining members, terminates the current epoch and starts the next one, reporting a view in which the failed nodes have been removed. State is transferred to any joining members. By the time the new epoch starts, we have a new value of *n* and a clean starting state, from which the system advances.

So what happens with our wedged update? The membership-changing protocol steps in to complete it. Thus it actually does run at *n-f* members, but in doing so we also terminate the prior epoch, start the new one, and treat the *f* unreachable members as faulty. Indeed, the membership service drops them from *every* group to which they belonged, and sets up a kind of software firewall that will prevent them from sending future messages into the system, should they be operational but temporarily inaccessible.

The effect is that a read can now be performed on any replica, because every replica has the full update history, up to the point in time when the read is scheduled. This is in contrast to a quorum scheme, where a read might access a replica that lacks some updates. In our world, such a replica is permanently inaccessible.

We see an illustration of this in (Figure 2). The figure doesn't show the membership service itself: we run it off the critical path in the background, on a small set of nodes, much as in the cloud computing services mentioned in Section 2. The membership service runs a consensus protocol to agree on each membership change, and is designed to never partition: if a network failure occurs, the membership service can only make progress in one of the subpartitions that result. We run the membership service on a set of nodes picked to ensure that it will always remain operational in the main body of the data center.

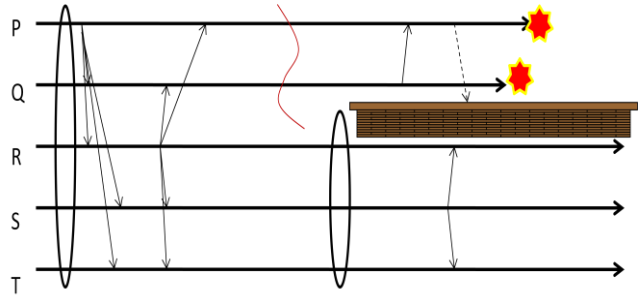


Figure 2: Replicated soft-state in our model. The execution looks like a state machine (hard-state) replication run until a network partition isolates nodes P and Q. The isolated nodes might still do some extra updates, but are prevented from talking to the other nodes (firewalled) and eventually shut down. Meanwhile, the main system reconfigures and resumes execution. A transient inconsistency arises (top right) but the firewall isolates P and Q until the reboot and rejoin.… but this means that some soft-state updates may have been dropped.

What is life like in the isolated set of nodes seen in the upper right in Figure 2? They were running normally until the partitioning event occurred. At that point they can briefly become inconsistent

with the rest of the system, but will rapidly (within seconds) realize that they are out of touch with the membership service. They then wedge themselves and, when the problem is repaired, reboot into clean states.

## 6. The Isis$^2$ System

The basic construct supported by the Isis$^2$ system is the *group*. The system is object oriented and just as there might be many objects, there will often be many groups. The system itself and the primary APIs (Figure 3) employ C# on .NET. We run on both Windows and on Linux, and there are cross-language APIs from a wide range of other languages. The assumption is that many user applications would be accessed by cloud-computing clients using the Web Services remote-invocation technology (the handler might run in a purely locally responsive manner in that receiving server, or might then trigger a more elaborate parallel query in which multiple servers collaborate to respond; we'll focus mostly on the former case). Thus Isis$^2$ applications might run in the outward-facing first tier of a cloud computing data center, or might implement tier-two services within the data center.

To join a group, the application must define the same type signature as do the other members, and if the group has a persistent checkpoint, the application must have file access permissions for the file storing that checkpoint. An optional security scheme associates a unique key with each group (256-bit AES). During periods when a group is inactive, the group key is stored with the checkpoint in the persistent backing store file. This allows us to leverage the file system access control mechanisms as the basis of secure-group access control: to join, a process must be able to obtain the key from this file. The same mechanisms are also employed for initialization of a joining member if a group is already live. We pick some existing member, and ask it to generate a checkpoint, which we then copy to the new member; it loads that checkpoint as its initial state.

There are two ways of performing reads. One is to just read the local state at any replica. The other option is to perform a parallel request in which all the group members participate to perform the requested action. This form of parallel query is done by multicasting to the group and having all the members send replies. Locking, if needed, is typically implemented by token passing. The membership of the group is available through a data structure we call the group view, and updates are reported through upcalls that are synchronized with respect to lock movement and update events. This allows the members to maintain consistent states (same data, updated in the same order, using the same updates), to tolerate failures, and to split work up, for example by by splitting a single request into $n$ parts, with the $r^{th}$ group member in the view handling part $r$ out of $n$ of the operation.

We could say a great deal more about the system, which is quite extensive and innovates in several ways relative to earlier virtual synchrony implementations, including our own Isis Toolkit. However, many of these are based on prior papers in which the specific mechanisms in question were first proposed. For example, Isis$^2$ uses IP multicast "safely", employing a technique from the Dr. Multicast [29] system to avoid overloading data center routers and NICs. The basic idea is to use allocate IPMC addresses to those groups with the largest membership and highest data rate, merging similar groups if the software filtering cost of doing so will be small enough. This often yields a 10:1 and sometimes a 100:1 "compression" of the number of IPMC addresses needed to support a given system. This gets us within range of a good story. As a next step, we limit the number of IPMC addresses to whatever the router can support without becoming overloaded. Groups that aren't assigned an IPMC address use point to point UDP messages to communicate (thus to send one multicast they send $n$ copies). Some cloud computing systems prohibit both UDP and IPMC; to run on those platforms, Isis$^2$ employs a TCP overlay, much like SplitStream [10].

IPMC isn't totally reliable, hence we need a scalable reliability mechanism. For this we use a structured gossip protocol, drawing on ideas from Bimodal Multicast [3] and Quicksilver Scalable Multicast [25]. The resulting protocol can handle large groups, large numbers of groups, and allows individual processes to employ whatever mixture of groups makes sense.

Of central importance for this paper are the mechanisms associated with non-durable soft-state updates and durable hard-state ones (and also with the corresponding forms of parallel queries, since these are initiated by a multicast to which the recipients respond in parallel). Here, *state* is whatever part of the state of a replicated object would be included into checkpoints and updated using multicasts. Isis$^2$ offers two flavors of multicast and parallel multi-query primitives. The non-durable FIFO-ordered versions are Send and Query. Variants we call OrderedSend and OrderedQuery can be used when concurrent multicasts will originate at multiple sources and a totally-ordered delivery is desired. The durable primitives are called SafeSend and SafeQuery; they offer precisely the same semantics as Paxos.

All of these primitives respect our extended virtual synchrony model: any object instance is consistent with any other replica of the same object, seeing the same updates in the same order. Locking is a user-specific matter: as noted earlier, some applications don't need locking at all; others use tokens to implement locking. The non-durable protocols are really bare-bones mechanisms. In a manner synchronized with the group membership, Send transmits a multicast using a single, unacknowledged (hence unreliable) IPMC, tunneling over TCP if IPMC is off limits or if the group just didn't get an IPMC address from Dr. Multicast. A decentralized background repair mechanism patches any gaps, and we use sender ordering to put messages into FIFO order by group and by sender, delivering them immediately upon receipt via upcall to the application. Send is completely asynchronous (the upcall at the sender occurs instantly, even before data is transmitted) and pipelined. And yet all of this is able to conform to the virtual synchrony model, because we coordinate membership changes using our membership service, and "flush" traffic in our groups prior to installing a new membership view.

If a failure does occur, durability is not guaranteed for Send and Query. The issue is as follows: for these cases, the sender transmits an update and applies it instantly, as do any receivers. But suppose this sequence were to occur within an isolated rack disconnected from the main data center. The main system will assume that these nodes are faulty, and will run its membership protocol, dropping the isolated nodes from any groups to which they belong. There was thus a brief window during which the isolated nodes saw and acted upon a soft-state update that the main system did not see. The effect is to "erase" the update (Figure 2, upper right), but only under this (unlikely) scenario.

But the situation could get worse: if the network fault heals quickly, inconsistency could "leak" from our isolated nodes when the network recovers and permits them to send messages into the main system. To see how we solve this, recall that when Isis$^2$ updates system membership, a consensus protocol runs. This membership service is designed to remain live in the main part of the data center and will not be accessible in the isolated partition.

```
using Isis;
namespace ExampleIsisApplication
{
   // Type signatures.  C# really ought to infer these from the context, but doesn't
   delegate void myLhandler(string who);
   class Program
   {
      int LOOKUP = 0;
      int UPDATE = 1;
      static void Main(string[] args)
      {
         // First create a group that includes just a single handler
         Group myGroup = new Group("some name");
         myGroup.ViewHandlers += (Isis.ViewHandler)delegate(View v) {
              Console.WriteLine("myGroup got a new view event: ", +v);
         };
         myGroup.Handlers[UPDATE] += (intArgs)delegate(string someStr, int val) {
               Console.WriteLine("My UPDATE handler got called with string {0}, new value {1}", someStr, val);
         };
         myGroup.Handlers[LOOKUP] += (myLhandler)delegate(string who) {
               Console.WriteLine("My LOOKUP handler was asked to look up {0}", who);
               myGroup.Reply(myGroup.GetView().GetMyRank());
         };
         myGroup.Join();
         // Now send some multicasts
         for(int n = 0; n < 10; n++)
                  myGroup.Send(UPDATE, "User number=" + n, n*100);
         // Finally, send a query
         List<int> ranksList = new List<int>();
         int nr = myGroup.Query(Group.ALL, LOOKUP, "John Smith",  EOLMarker,  ranksList);
         string reps = "";
         foreach (int rep in ranksList)
                reps += " " + rep;
         Console.WriteLine("Got {0} replies and they are {1}", nr, reps);
      }
   }
}
```

Figure 3: A sample of the code style supported by Isis$^2$. This nonsense example creates a group that implements an update handler and a multi-RPC handler, and reports new membership views, all by printing to the console. The coding style is similar to the way one builds a GUI with handlers for events like mouse clicks. The C# delegate() { … } construct is used to declare an event handler inline.

Thus, any nodes in the small partition, if they survived the switch outage, will quickly discover that they are isolated (within seconds) and shut down. We're working with soft-state, so when they restart after the problem that isolated them has been resolved, they do so in clean states. The only risk of a leak arises during the brief after the membership service in the main system has dropped these nodes as faulty, but before they notice the condition themselves and wedge, preparing for some future reboot. During this period, if communication suddenly were restored, the isolated nodes might get a chance to send messages to the main system, which would perceive these as coming from the dead.

One can see this in Figure 2. The isolated nodes and the main system diverge starting when the main system declares the isolated nodes as faulty, as evident on the upper right: some messages are exchanged within the isolated pair of nodes (P and Q). The figure illustrates our solution to the risk this creates: we activate a software firewall; it blocks messages from a node that has been dropped by our membership service. Thus, an inconsistent node can't talk to the main system during the brief period before it discovers its isolated and inconsistent state and reboots.There is a second case to consider, because our firewall only applies to messages sent using the Isis$^2$ primitives. Thus, one could imagine situations in which the isolated, inconsistent nodes manage to talk not to other Isis$^2$ nodes, but to an outside user. In Section 7 we'll introduce an additional primitive, Flush, targeted to resolving this specific scenario. The application invokes Flush before communicating to the external user, and this delays the caller until any pending Sends have become durable.

In effect, the Isis$^2$ developer is offered a choice. One option is to use Send, a protocol designed for soft-state updates. Doing so poses the risk that certain sequences of events associated with partitioning might deliver an update but then lose it. This should be rare, and we limit the risk of inconsistency leaking to the main system with our software firewall. As a further protection, the developer can call Flush prior to interacting with external users. Alternatively, the developer just treats the state as hard and uses SafeSend, a protocol that has the same properties as Paxos.

Claims that a protocol is like Paxos requires qualification, because Paxos has several versions. The variant most like SafeSend is "general" Paxos with a leader and with dynamic membership. In our implementation, a parameter $\Phi$ tells Isis$^2$ how many members of the group should acknowledge receipt before delivery occurs

(how many are *acceptors*). All group members deliver all messages (hence, all are *learners*). The value of Φ must be large enough to guarantee durability; on platforms that don't provide physical layout information, we use Φ=*n*. Readers interested in learning more about "virtually synchronous Paxos" will find details in [5].

How will all of this scale? Hopefully, the reader will easily see why the non-durable Send scales well: a user-initiated multicast translates directly to an IPMC, and because we use Dr. Multicast, IPMC should be nearly lossless and extremely fast. A little background chit-chat over a gossip protocol deals with any loss that still occurs, but normally, no action is needed: multicasts are almost always received successfully and delivered via upcall to the application instantly. Reads are purely local, so those introduce no delays at all. Most locking can be avoided. Add this up, and we have an extremely scalable, locally responsive service with strong consistency. Moreover, while Send (without a Flush) doesn't guarantee durability, the sequence of events needed to violate durability should be very rare.

In contrast, SafeSend scales less well. As we increase the number of "acceptors" (and especially if we are forced to treat all members as acceptors) the delay before we can confirm that an update is safely deliverable grows steadily. In practice, developers who are certain about the application layout would limit the number of acceptors to some small number, perhaps 3 or 5. However, even so, SafeSend will be sharply slower than Send, and for developers who lack the needed layout information, the only choice will be to keep hard-state groups small.

We believe that applications that need to support what Vogels calls "ambitious" scalability would thus be driven towards the faster yet weakly-durable Send: the degree of performance difference we'll see in Section 9 is compelling. In contrast, for any genuinely hard state the developer will be forced to employ small replication factors: just enough copies to avoid loss in the event of a failure. These observations echo Brewer: one scales soft-state, and this eases the load on hard-state services, which is good because hard-state services can't scale nearly as well.

## 7. CATEGORIES OF SOFT-STATE

We now encounter a subtle concern, relating to the question of when one needs to use Flush. To explain the issue, it may be useful to think of soft-state in terms of three subcategories:

1. Pure soft-state without hard-state dependencies. Examples include queue-lengths in a replicated load-balancing subsystem. Such state has no direct connection to hard-state.

2. Soft-state that caches or otherwise mirrors underlying hard-state. Any value the soft-state exhibits is one the hard-state took "first". This kind of soft-state depends on hard-state.

3. State that has no hard backing storage and yet for which the user expects that updates not be lost. A related case arises if hard-state somehow has values that are dependent upon (and that should be consistent with) a prior soft-state update.

We're proposing to use SafeSend (Paxos) when updating hard-state and Send for the first two kinds of soft-state. For the first case, the mapping to Send is evident. For the second, it is still easy to see why durability isn't needed: if a cache is cold after a node restarts, we'll just reload it from the hard-state. In effect, soft-state only takes on states that hard-state took on first, so our only issue here is one of fault-tolerance[2].

But think about case 3: suppose that a service that did a series of soft-state updates is about to send a result to an external user, and in some sense, the reply only makes sense if those soft-state updates are retained. For performance and scalability reasons, we would prefer that the developer to treat the state in question as soft-state and update it with Send. Yet if these updates were lost the end-user would perceive the system as inconsistent. Accordingly, as mentioned earlier, Isis[2] includes an additional primitive, Flush, which the application should invoke *after* updating this kind of soft-state and *before* interacting with the external user or updating hard-state. Flush delays the end user response so that it won't be sent until stability for any prior non-durable multicasts has been achieved. Thus, by the time the external user sees anything, all updates on which that reply depended have become durable.

As seen in Figure 4, flush applies to the updates known at the time the primitive is invoked. New incoming soft-state updates will still be delivered, and can be applied. This poses no difficulties because by the time Flush is invoked, the application has already computed the value that it wants to transmit to the external user.

Flush breaks local responsiveness, and Send+Flush are slower than SafeSend. However, applications that do a series of Sends and only then call Flush will still be mostly locally responsive. This is encouraging because such patterns will be common. The BASE methodology specifically encourages the developer to create pipelined operations that execute as a series of steps: a series of Sends. The effect is that any single call to Flush will have its cost amortized over a potentially large number of soft-state updates.

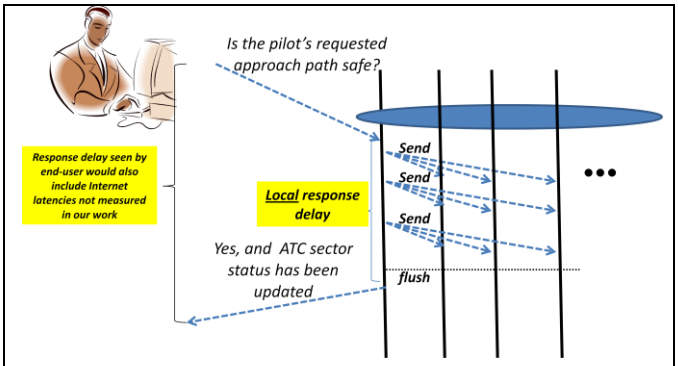


**Figure 4: Use of Flush, after a series of weakly-durable Send operations, prior to responding to an end-user. The extent to which the system is *locally responsive* will be a major contributor to the perceived response delay of the service.**

In summary, our mostly-durable soft-state update primitive can be used "as is" in many settings. If an application might be at risk replying to an end-user on the basis of non-durable updates, by using Flush the application can delay the reply until any soft-state updates on which it depends become durable. But one wouldn't call Flush after each and every soft-state update: by doing so at the last instant, we achieve a highly amortized, parallel execution

[2] The platform makes it very easy to ensure that a role, such as forwarding hard-state updates into a soft-state cache, will be performed fault-tolerantly. Thus while this isn't an automated function, it is straightforward to implement.

that has most of the advantages of local responsiveness, but offers absolute safety. For hard state we offer SafeSend, our version of Paxos, which won't deliver an update anywhere until durability is certain. The model is identical to Paxos, but our implementation has an important advantage over other ways of implementing the protocol: the virtual synchrony group membership property allows us to eliminate quorum reads. Thus, when Send is employed we support massive scalability and a big speedup., but even with SafeSend, we achieve improved performance compared to prior state machine replication solutions.

# 8. APPLYING THE METHODOLOGY

Our model brings benefits both for soft-state and hard-state: for soft-state, we get correct, locally responsive (or almost entirely locally responsive) behavior and are also able to scale out with very high performance; for hard-state the scalability story isn't quite as good, because each update must be acknowledged by an adequate set of receivers before any process can apply the update. To have a successful high-assurance cloud story, we'll need to find lots of soft-state and lots of read-only operations. If a typical high assurance application were to be write-intensive with mostly hard-state, there might not be enough opportunities to benefit from our fast but non-durable update primitives.

With this in mind, let's ask how much state is likely to be soft in a high-assurance application, and how consistency would benefit the developer who works with soft-state. We'll use an air traffic control scenario, but a similar analysis would be possible for financial, health care, or power grid control scenarios, and we believe that conclusions would be similar.

An air traffic control system is a very large infrastructure with a safety critical role, and this extreme concern about safety tends to have a shadow effect, forcing a safety analysis even for subsystems that might not have obviously critical roles. Clearly, any instructions given by the ATC system (whether by human controllers or automated services) must be safe. But to get these decisions right it becomes important to know which controllers will handle each stage of a flight, so that if one controller needs to coordinate a decision with another controller, they can easily contact one-another. That leads to the question about whether we need to provide consistency for mundane details about which controllers are on duty in each flight center, which in turn might lead to questions about who is taking coffee breaks, and who the temporary backup sector assignment was given to, etc.

In fact, we clearly do need consistency even for these kinds of obscure aspects of ATC system state. For example, suppose that an ATC system prioritizes flights having many passengers with tight connections over flights that have ample fuel and are early or on time. Which parts of this information are safety-critical hard state and which are soft-state? Clearly, the actual decision to allow a plane to start its final approach and landing sequence is a safety critical action, as is any other decision that routes a plane into a contended-for portion of air space. The "tight deadlines" data looks non-critical. But one might not want to jump to the conclusion that such data can safely be handled in inconsistent ways. Even if doing so didn't put lives at risk, it could easily lead to angry disputes (e.g. if priority is given to the wrong flight).

Seen this way, many kinds of very minor information could take on roles that require consistency. In today's interpretation of CAP and BASE we would probably view all of this state as hard-state and conclude that we probably can't build a scaled-out cloud-hosted ATC system because we won't be able to replicate the underlying sate in a safe and high-performance manner.

By bringing consistency to soft-state, we can potentially tackle that performance-limited scaling step. True, we shouldn't pretend that soft-state is somehow hard-state, and we need to be careful of the implications of mostly-durable updates; errors here could compromise safety. Many of the state examples just listed would fall into case 3 in our dichotomy from Section 7: a call to Flush will be required before interactions with the external world. Yet with that extra step, treating them as soft-state permits us substantially improved scalability and performance.

Given a consistent way to replicate soft-state, one can suddenly see many forms of soft-state here. The data about tight correspondences, for example, starts to look like soft-state, as would a great variety of other state. And some of this other state might be surprising. Consider the GPS readings showing the current locations of planes. While the plane itself and the primary radio that tracks it will need to maintain a durable record of this data (a small hard-state record), the rest of the ATC center may be able to work with aggressively replicated soft-state, very much along the lines of case 2 in Section 7: coherently cached data backed by a durable hard-state record.

Within complex systems, many kinds of information have this same character. For example, consider internal status data: which controller is currently seated at which console? Which nodes are running the main ATC system services, and what IP port numbers are they listening on? Which services are currently online? If we can "trust" soft-state, then all of these kinds of data are soft.

Notice that we're doing just what Brewer first urged when he introduced the idea of a soft-state / hard-state distinction in his first talks on this subject. We're using hard-state to record the permanent record of key data, but then creating a much more scalable soft-state front-end to that data. Yet whereas BASE takes that last step of giving up on consistency, we don't need to do so.

Without dragging out this analysis, the reader will see that we're identifying quite a bit of potential soft-state, and that we wouldn't have been able to do so had we not been in a position to guarantee consistency for soft-state replication. And while we're planning to use a relaxed update durability guarantee for this state, doing so won't introduce risks because we are able to offer a single, all-encompassing consistency model that spans both soft-state and hard-state, permitting the developer to reason carefully about application correctness.

# 9. EVALUATION

We undertook a series of experiments aimed at comparing the scalability and raw performance of Send, Send followed by Flush, and SafeSend, using 100-byte payloads. Our experiments ran on a lightly loaded 78 machine cluster, each with dual-socket Intel Xeon X5650 Nehalem processors having a combined 12 cores and running a Linux 2.6.18 kernel, giving a total of 936 cores. These machines are connected through a 1Gbps (Intel 82574L) Ethernet.

Any data center built with multicore servers will run multiple application instances on each machine, hence we started by asking how this form of application stacking impacts performance of the core mechanisms used in our protocols. To evaluate this, it makes sense to look at a simple multicast, and at a simple request-reply protocol. Figure 5 evaluates first the latency from when a Send is initiated to when it is delivered, and then the total cost for a request/reply "phase", first for an application with 1 copy per node, and then again with 12 copies per node. Throughout, each individual data point reflects a run that performed 20 of the indicated operations, and we repeated each experiment 10 times. We omit the standard deviation for reasons that will become clear.

| N | Delivery latency 1/node | Delivery latency 12/node | Multi-RPC 1/node | Multi-RPC 12/node |
|---|---|---|---|---|
| 24 | 1.89 ms | 2.15 ms | 21.88 ms | 7.59 ms |
| 36 | 2.22 ms | 2.01 ms | 31.52 ms | 8.38 ms |
| 48 | 2.28 ms | 2.07 ms | 34.07 ms | 11.65 ms |
| 60 | 2.21 ms | 2.63 ms | 33.63 ms | 14.76 ms |

**Figure 5: Impact of stacking several application copies / node.**

Our smaller experiments, below, used one copy of our application per node, but the N=800 experiments stack 12 copies per node. From Figure 5 we can see that stacking is having both positive and negative impacts: stacked latency figures are comparable to the unstacked ones, but the stacked multi-RPC experiments exhibit reduced time-per-phase, presumably because intra-node communication is faster than inter-node communication and less prone to packet loss. As a result, we recommend caution in comparing Figure 9 with our other graphs: Figures 6-8 all use one copy of the application per-node, while Figure 9 was run with 12 stacked copies per node. As a side remark, we should perhaps mention that we've experimented with more than 12 copies per node. Variance rises, and we solid evidence of overload and contention. For brevity, we omit that data here.

Figure 5 is also interesting because it gives some sense of the actual cost of request/multi-response "phase", in which an IP multicast elicits replies from all receivers. This pattern has an important role in both Flush and SafeSend. In the case of Send followed by Flush, a Flush waits for termination of the out-of-band acknowledgement protocol used by Send to confirm reliability: an instance, then, of the scenario measured in Figure 5. In SafeSend, that same pattern arises during the initial send phase of the protocol, which must complete before SafeSend's second-phase message, in which delivery occurs.

Our next experiment explored the impact of varying the value of Φ in SafeSend as a function of group size. On deployments of up to 65 nodes, we ran SafeSend with Φ=1, Φ=3, Φ=N/2 (H), and Φ=N. Curves plot average latency as a function of number of members in the group. The figures show both these average latency curves, as well as scatterplots of the underlying latency measurements (across 10 runs with 20 request / replies each). Recall that without knowing how applications were deployed (stacked), an application seeking real safety must use Φ=N.

We included the Φ=H case as an optimistic midpoint for developers who run with large N, and yet have little control over stacking, reasoning that the "odds" of durability violations would be low, and that the last members to acknowledge receipt might be delayed by a scheduling event, such as garbage collection.

Figure 6 shows that if one looks only at its *average* delivery latency, SafeSend scales fairly well for small values of Φ. Note, however, that variance is high throughout all our runs, especially when we work with Φ=H or Φ=N (this is also why we omitted the standard deviation in the table for Figure 5). Φ=N gives the worst performance, by a substantial margin, but high variation in latency is seen for all deployment sizes and all choices of Φ>1 (and there is even one long delay with Φ=1, N=32). Thus we reach two initial conclusions. First, developers working with SafeSend, or with Paxos, would have a strong interest in using a small value of Φ (a small "acceptor" set) even if N, the number of receivers (the number of "learners") is large. We informally surveyed prior papers on Paxos performance and confirmed that indeed, all published results seem to do this; safe or not, Φ=3 is typical. But we also see that large variance is a concern on the cloud.

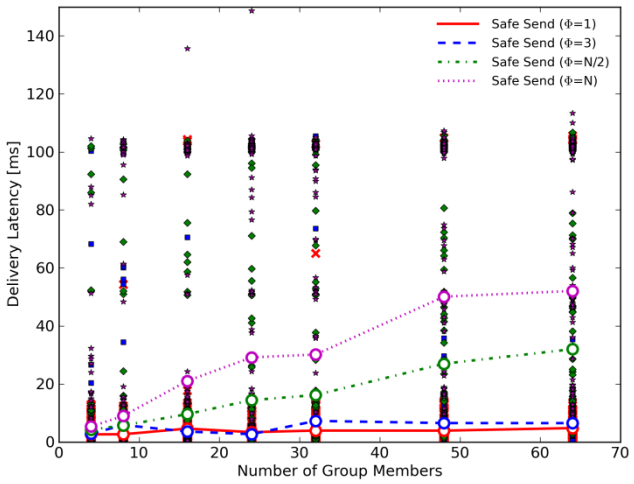


**Figure 6: SafeSend for various values of Φ.**

These large delays would have a substantial impact on application responsiveness (e.g. in a scenario like the one from Figure 4, but with Send and Flush replaced by SafeSend). The problem here is that unlike with Send, where an update can be applied by the sender as soon as it is initiated, without any wait for remote processes to respond, SafeSend can't even apply an update at the sender until Φ-1 remote members have acknowledged the message. Thus, just as with a quorum read, the sender won't know the "state" of the data being updated and incurs this full delay. In contrast, an application using Send would see this delay as purely "background" cost: for Send, the sender does a local update instantly, hence the delivery latency is entirely a question of how long it will take before the *other* group members catch up with the state at the update sender.

Overall, Send has remarkably low delays in comparison to SafeSend. Yet notice that even Send has a slow delivery now and then, for example in the 8 and 32-member cases, where we see isolated instances of Send latencies running at around 60ms. Thus long delays can occur with both protocols (just much more often for SafeSend), and require some explanation.

In discussions with our colleagues, many jump to the conclusion that the choice of C# as the implementation environment (here, under Mono on Linux) would trigger performance issues. Now that we're in a position to study the real system and its performance, those easy explanations are turning out to be mostly wrong. On the one hand, C# is a fast, compiled language. We chose it because we found that its type-safety, reflection features and utilities matched to our needs; many developers would do so as well. Obviously, C# does extensive memory management (and hence requires garbage collection), and its reflection features are slow in some situations. Yet to assume that these language features cause the extreme performance variance seen with SafeSend, yet not for Send, misses a deeper insight.

First, we have evidence that the issues are elsewhere. C# has several options for controlling garbage collection, and we use them. Triggering the collector very frequently (or very rarely) had no impact on the performance variation observed here. More serious and harder to control for is the scheduling of threads: Isis[2] makes extensive use of multithreading, with several background tasks that run in threads of their own, 2 threads per process group created by the user (one for asynchronous sends, one for asynchronous receives), and 2 more threads per physical IPMC address in use (same roles). We can and do use thread priorities,

but that represents is a coarse-grained control knob: mostly, it lets us prioritize receiving incoming packets over sending new ones. Finally, C# (like any language) has profilers. Our bottleneck is communication; Isis$^2$ itself isn't doing much computation.

In fact, the performance-limiting factor, and the source of high latency "runs" of SafeSend, seems to be packet loss, triggered when a burst of messages arrives at some single process faster than it can receive them off its incoming sockets, or when the process is asleep because of O/S and .NET scheduling delays, with the latter perhaps being the "main" issue. Notice that with Send, high-latency events are rare: we see the occasional single packet loss, and incur a 50ms or so retransmission delay when that occurs. SafeSend, in contrast, is at risk of multiple "chained" losses because of its multi-phase structure, yielding a broad spread of latencies that correspond to loss in the first phase, in the reply phase, in the second phase, etc. Our data thus supports the view that delays triggered by packet loss explain the high variance.

In the cloud, a constellation of factors conspire against steady real-time behavior, and events such as O/S packet loss are astonishing hard to prevent. We experimented with very small retransmission delays and larger socket receive buffers; both *increased* loss rates. Quick retransmissions just send duplicates to the stalled process, crowding out real data. Increasing the socket buffering puts stress on kernel memory resources, increasing kernel-level packet drops. Thus, the sharing of resources at the center of cloud-scale efficiency is also an obstacle to predictable, steady performance and fast replicated data updates.

We now arrive at the first of the central questions posed by our work. In Figure 7, we place SafeSend and Send side by side, using the "very safe" Φ=N case for SafeSend. As is evident, the decision to work with a durable update has substantial performance impact, and the scalability of the resulting solution is relatively poor in comparison with that for Send. Indeed, one can barely see the latencies for Send: they are flat and run at roughly 2-4ms irrespective of group size, while SafeSend latencies rise to 50ms but also vary wildly, reaching as much as 120ms for quite a few of our runs. Once again, this variance is almost certainly a sign that SafeSend was triggering packet loss at the node that leads the protocol, which is be flooded by N responses to each multicast in near synchrony (with IPMC in use, as was the case here, the receivers receive each packet within a spread of at most a few tens of microseconds). Thus, our hypothesis is that as group sizes group, the sender is overwhelmed, drops packets, and since delivery occurs in phase 2, latencies soar.

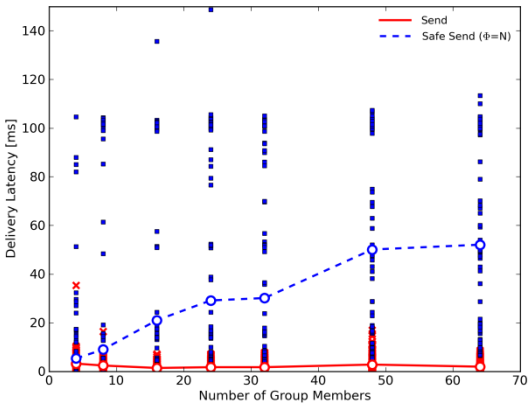


**Figure 7: Comparison of Latency to Delivery for Send and for SafeSend in the Φ=N (all members) case.**

In Figure 8, we look at the bottom line impact of these choices for response delay of the kind shown in Figure 4. Recall that our fundamental goal is to support *locally responsive* services that are elastic. In this experiment, we measured response delays in an application that sends a series of updates using Send or SafeSend, then calls Flush in the case of Send, precisely as in Figure 4. Then, the application responds to whatever externally generated request initiated the computation (note that for forms of soft-state that don't require a Flush, the response delay would be near 0). The measured response delays are shown on a log scale because the range varies so widely.

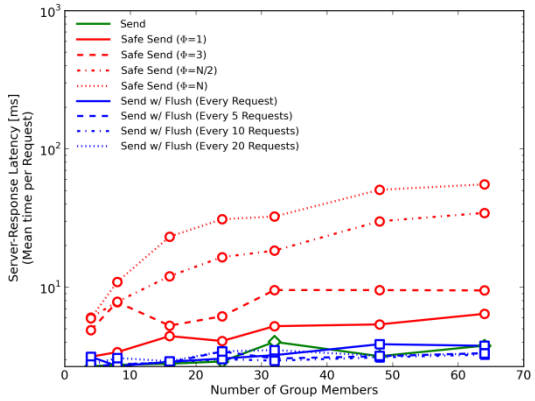


**Figure 8: Delay before end-user response can be sent for various update options (log scale)**

Send, even with a Flush, outperforms all SafeSend configurations: Send is fast, and scaling extremely well. Looking just at average response delays (ignoring variance, to which we'll return momentarily), SafeSend is fairly fast for small values of Φ, less so for Φ=N/2 (shown as "H") and scales poorly for Φ=N. From this is seems clear that Send achieves our basic objective: elasticity and locally responsive behavior. From Figure 8 alone, one might also conclude that SafeSend also scales "relatively" well with small Φ. But variance lurks as an issue under the surface here: we didn't graph standard deviation in Figure 8 because variance was so wide that the distribution really isn't one that can appropriately be characterized by a mean/s.d. value of the sort so common in systems experiments. The usual alternative is to show the actual distribution of data points, but for Figure 8, with so many cases, that would create a confused jumble. But look at Figure 9, where we include a scatter plot, showing delivery latency in a much larger configuration.

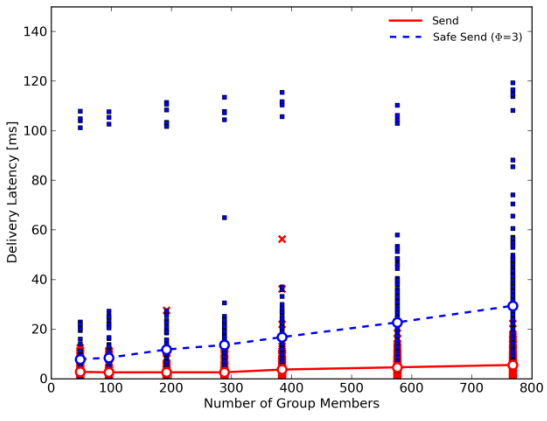


**Figure 9: Latency before an update can be applied in very large groups using Send (red) and SafeSend with Φ=3 (blue).**

To create Figure 9 we tackled much more more aggressive scenarios (and we're not done: later in 2011 we hope to report on experiments at ten to fifty times this scale). Here we've gone with Φ=3 for SafeSend: the value that seemed most plausible in Figure 7, despite its implicit assumption that the designer would need to control application mapping to hardware on the cluster. Now we measure delay before delivery occurs (the metric employed in Figures 5 and 6) but with as many as 800 group members, stacked 10 per node on a total of 80 machines.

Two aspects are notable: first, Send is scaling with essentially flat latency even in these larger groups, and second that SafeSend is not only slowing down, but continues to have very high delivery variance, often exceeding 100ms. There are also a few cases in which we see these kinds of delays with Send As noted earlier, these are apparently associated with packet loss associated with the cloud-computing "model", not by our use of C# or by its occasional need for garbage collection. Heavy communication loads burden the kernels on the target systems, which apparently drop packets more frequently, and there is little that we (or any application) can do to control such problems on the current generation of cloud operating systems and runtime environments. Send seems less prone to this problem: it runs with a leaner footprint, messages linger in the system for less time, and fewer messages are required to complete each run.

In summary, our experiments confirm that for soft state updated using Send, we achieve striking scalability: the sender itself is able to be locally responsive even if it pauses to do a Flush prior to replying to the external user, and the soft state replicas at other nodes will be far more current than in current soft-state update schemes, where updates may require many seconds or even minutes to reach replicas. With our implementation, though, even hard state updates scale better than one might have expected purely from reading the data center literature. While hard-state updates are sharply slower than soft-state updates and exhibit high variance, it would not be out of the question to consider using hard-state even in a scaled-out service. Moreover, although not measured in these experiments, recall that because our system reconfigures when a member crashes, SafeSend is able to update every replica of a replicated object, eliminating the need for quorum reads or quorum writes.

The main negative story in our data reflects the extremely complex execution scenarios that we're working with and their somewhat mysterious interactions with response delay. In the most aggressively-scaled cases, we had 10 copies of our application running on each multicore machine, perhaps as many as 200 threads per machine, and all sorts of things going on, including (but not dominated by) C# garbage collection. The effect of this was to create enough scheduling delays that socket overflows occurred, significantly delaying response.

It is interesting to realize that even though cloud computing systems are aimed at supporting locally responsive services, the long-delay events for Send in Figure 9 are in part caused by the way that cloud systems map application to nodes so as to promote efficiencies of scale. In particular, notice that long-delay events seem to be less common in Figure 7, where we ran just one instance of our Send application on each node, and more common in Figure 9, with 10 Send instances per node. Thus, the kinds of things cluster managers do to keep the cloud busy, such as application stacking, are also contributing to erratic scheduling, which in turn increases the risk of socket overflows that result in loss. With SafeSend at larger (safer) values of Φ, the issue becomes extreme.

All of this suggests that if cloud management systems were to expose more information about the application-to-node mapping, and to permit a bit more user control over that mapping, users focused on responsiveness would gain much better performance. Moreover, since the cloud platform will end up running the same number of application instances, the cost to the cloud operator of offering such options wouldn't be particularly high (the main effect is that when many VM instances are stacked on one machine, the VMM can sometimes share pages between the instances and the file system can sometimes share file system caches across the instances: useful features, but here we're seeing their downside).

As a final comment, we note that prior work with virtual synchrony generally failed to scale beyond 50-100 members per group. Our data suggests that the new Isis$^2$ system isn't having any trouble scaling far beyond this: we're already exploring scenarios ten times larger than that, and see absolutely no evidence at all that our 800 member scenario is approaching any kind of limit. Brevity prevents a detailed exploration of precisely why Isis$^2$ scales so much better than the older Isis Toolkit or other prior virtual synchrony systems of which we're aware, but the main difference may simply be that Isis$^2$ was designed with scalability as a main goal, and includes batch-style APIs to add many members to a group in one operation, or to remove many, or to delete a group when the application using it shuts down. The main uses of past versions of virtual synchrony systems involved modest levels of server replication. Lacking an incentive to support batch APIs, they did everything one member at a time.

# 10. RELATED WORK

We believe that we are the first to suggest that the "real" point of BASE [26][32] is to achieve local responsiveness. We are not aware of any prior work that traces the costs of consistency as used in CAP [8][17] to durability, or that shows how durability can be relaxed for strongly consistent soft-state updates.

There has been debate around CAP and the possible tradeoffs (CA/CP/AP). Relevant analyses include Kossman's [19] [20] [7], and Abadi's [1] discussions of this topic. Database research that relaxes consistency to improve scalability includes PNUTS [11], the Escrow transaction model [26] and Sagas [15]. At the other end of the spectrum, notable cloud services that scale well and yet offer strong consistency include GFS [16], and Zookeeper [18]. Roy Friedman explored relaxations of virtual synchrony in support of highly scalable real-time telecommunication switches [14]. Relaxations of consistency to improve scalability of Byzantine Agreement were studied in [31]. Of course, this is just a partial list; a comprehensive survey would require a paper in itself. Papers focused on performance of Paxos include the Ring-Paxos protocol of [23][24] and the Gaios storage system [6].

The work presented here employs a model that unifies Paxos (state-machine replication) with virtual synchrony [5]. Other mechanisms that we've exploited in Isis$^2$ include the IPMC allocation scheme from Dr. Multicast [29], the gossip-style multicast reliability idea from Bimodal Multicast [3], and a tree-structured aggregation mechanism first used in QuickSilver Scalable Multicast [25].

# 11. CONCLUSIONS

An "ACI and mostly D" model offers a way to accept the CAP theorem and the premises of the BASE methodology and yet obtain solutions that are scalable, strongly consistent, and fast. Indeed, if IPMC is available, our new approach may offer far better performance and scalability for update-intensive uses that

today's cloud-computing solutions. This is not to say that the existing BASE approach, with eventual consistency achieved through a variety of convergence techniques, won't continue to play an important role. But we believe that a genuinely strong and yet scalable consistency option could eliminate much of the need for the more extreme steps in the BASE process. Moreover, for applications where high assurance is a requirement, our approach allows the developer to scale out in a BASE-like manner without needing to implement potentially costly or complex repair mechanisms to overcome inconsistencies that might arise in a standard BASE development approach.

Our approach isn't the whole story. Some high assurance applications will need enhancements to the Internet, others may require completely new security standards, and there will also need to be work on ways of protecting against errors that occur on the client side. Moreover, our consistency model may not suit all uses (for example, we don't address the needs of applications that require Byzantine fault tolerance). But we believe that Isis$^2$ targets an important and large class of applications that are already migrating to the cloud, and that these represent just the tip of a coming wave.

## 12. SOFTWARE AVAILABILITY

The Isis$^2$ platform will be available for download from our web site by mid 2011, under FreeBSD licensing.

## 13. ACKNOWLEDGEMENTS

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

## 14. 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] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. 11th ACM Symposium on Operating Systems Principles, Dec 1987.

[3] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu and Y. Minsky. Bimodal Multicast. ACM TOCS, Vol. 17, No. 2, pp 41-88, May, 1999.

[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] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. SIGOPS Oper. Syst. Rev. 37, 5 (October 2003), 298-313.

[11] B. Cooper, et. al. PNUTS: Yahoo!'s hosted data serving platform, Proc. VLDB 2008, 1277-1288.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. ACM OSDI 2004.

[13] G. DeCandia, G., *et. al*. Dynamo: Amazon's highly available key-value store. In Proceedings of the 21st ACM SOSP (Stevenson, Washington, October 2007).

[14] R. Friedman, K. Birman. Using Group Communication Technology to Implement a Reliable and Scalable Distributed IN Coprocessor. Proc. TINA 2008.

[15] H. Garcia-Molina and K. Salem. SAGAS. SIGMOD 1987, 249-259.

[16] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. 1$^{9th}$ ACM SOSP (Oct. 2003), 29-43.

[17] 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

[18] F. Junqueira; B. Reed. The life and times of a ZooKeeper. Proc. ACM Symposium on Parallel Algorithms and Architectures (SPAA),Aug. 2009

[19] D. Kossman. Keynote talk, Eurosys 2011 (April 2011).

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

[21] V. Kundra. Federal Cloud Computing Strategy. http://www.cio.gov/documents/Federal-Cloud-Computing-Strategy.pdf. February 2011.

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

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

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

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

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

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

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

[29] 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.

[30] R. van Renesse, F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. OSDI 04.

[31] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, P. Maniatis. 2008. Defining weakly consistent Byzantine fault-tolerant services. In Proc. 2$^{nd}$ Workshop on Large-Scale Distributed Systems and Middleware (LADIS), 2008.

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