A Case for Fluid Replication

Brian Noble, Ben Fleis, Minkyong Kim
University of Michigan
Ann Arbor, MI
{bnoble,lbf,minkyong}@umich.edu

Abstract

The performance of distributed services is becoming increasingly variable due to changing load patterns and user mobility. The two approaches to this problem, cluster-based, scalable services and peer replication, solve only part of the problem. Cluster-based services deal only with end-service variability, while peer replication compromises safety and precludes the ability to offer bounds on consistency of updates. We propose *fluid replication*, automatic creation of replicas where and when they are needed, as a solution to this problem. In this paper, we present our mechanisms for finding replica sites, balancing consistency and performance, and maintaining client consistency when changing replicas.

1. Introduction

As distributed services scale, their performance becomes increasingly unpredictable. There are three reasons for this. First, aggregate user interests tend to change rapidly and unpredictably, placing varying demands on services. Second, changing patterns of demand in the underlying network also lead to variable performance, even if the demand for the service itself has not changed. Third, users are becoming increasingly mobile; as they move, the costs of accessing their set of preferred services change.

Unfortunately, users have little tolerance for variations in service time [33]. To provide more consistent response, a service must address each of these causes. One way to do so is through *replication* of the service and its data in response to changing user demands and resource availability. This can increase system capacity, decrease dependence on congested resources, or both.

Manual administration of replicas is clearly not up to the task, though it presents useful heuristics one might employ. For example, one might place replicas on either side of a slow network link [31]. Unfortunately, people are simply too slow to keep pace with the rapid changes in demand for resources. Furthermore, administrators often dedicate resources to transient trouble spots; as load decreases, those resources are not reclaimed.

Cluster-based, scalable services address some of the sources of performance variability [6,8,26]. In this approach, services are deployed on a tightly coupled cluster of machines. When a service discovers heavy load, it replicates itself elsewhere on the cluster, and the entry point to the cluster redirects requests for load balancing. This technique is effective when the end service is the bottleneck, but does not address variability caused by changing network demands or user mobility.

In contrast to this server-based approach, autonomous, peer-replicated systems place the responsibility for replication with clients [13,18,36]. Each client caches a subset of the data in which it is interested, and operates on that local cache. When peers encounter one another, they exchange updates.

Peer-replicated systems address network- and mobility-induced variations in performance, but introduce other problems. Update propagation depends on user mobility patterns, and these are outside of system control. Therefore, one cannot offer bounds on update convergence or consistency. Further, when a client overruns its local caching resources — a possibility on lightweight, low power devices — it must fall back to its original, remote service, which may be expensive to reach. Finally, client machines are necessarily less trustworthy than services [15], as they do not enjoy the same administrative diligence. Hence, updates that are stored only on clients are more vulnerable to loss compared to those stored on a server.

Our goal is to provide the safety and bounded consistency of server-based approaches with the performance and efficacy of client-based schemes. We propose *fluid replication* — the automatic creation of replicas where and when they are most needed — as a way to provide these properties. In fluid replication, clients monitor their observed performance in interacting with the service. When performance becomes poor through increased network load or client mobility, a replica is created to reduce the dependence upon the poor network path. Central to this approach is the WayStation: a service node on which a replica may be created. We view fluid replication as a complementary approach to cluster-based services; the former deals with network-induced performance problems, while the latter addresses end-service problems.

Fluid replication has the potential to provide several tangible benefits. It enables the automatic allocation of network resources, balancing them to offered load at fine grain through local, autonomous decisions. It hides performance problems endemic to large scale systems with a wide range of capabilities and many mobile users. These benefits accrue without unduly sacrificing consistency, leaving updates vulnerable to loss, or limiting the ability to offer bounds on update propagation.

There are several challenges that must be met to provide fluid replication. First, one must have a way to decide that a new replica is needed, and a mechanism to select a WayStation on which to host it. Once a replica is established, one must balance the consistency and performance seen when using that replica. Finally, one must reclaim WayStation resources in a way that preserves the consistency properties that clients expect.

2. Impact of Networking Costs

To demonstrate the potential impact of increased networking costs on common distributed services, we examined the impact of latency and bandwidth constraints on a small distributed file system workload. In this experiment, an NFS [30] file server stored a small source tree that was compiled by a client. These two are connected to each other through a host capable of network trace modulation — the delaying of packets according to a simple model of network performance [25]. The client compiled the source tree, storing objects on the server, over four different networking scenarios: unmodulated 10Mb/s Ethernet with negligible latency, 10Mb/s with 20 ms latency, 100Kb/s with negligible latency, and 100Kb/s with 20 ms latency.

	10Mb/s	100Kb/s
<1 ms	193 sec (3.0)	303 sec (3.2)
20 ms	986 sec (2.2)	1071 sec (2.8)

Table 1: Network Impact on NFS Benchmark

The results of this experiment are shown in Table 1. Each cell shows the average running time over five trials, with standard deviations reported in parentheses. NFS was specifically designed for local area networks, and it is not particularly frugal in its use of them. Even so, the impact of moderately poor networking performance, particularly latency, is severe. Rather than forcing a redesign of NFS to cope with such environments, fluid replication admits the possibility of eliminating poor network performance from the critical path.

3. Monitoring Network Performance

Fluid replication is a reactive mechanism. It monitors the performance of remote services, and when it becomes poor, the system reacts by creating a replica in a more advantageous location. Service time might be judged unacceptable for two reasons. First, server-side delays may increase due to server load. In that case, the best location at which to place new capacity is near the overloaded service; existing cluster-based solutions do precisely this. Second, the costs to communicate with remote services might increase. In reaction to such increases, the affected client must create a *remote replica*.

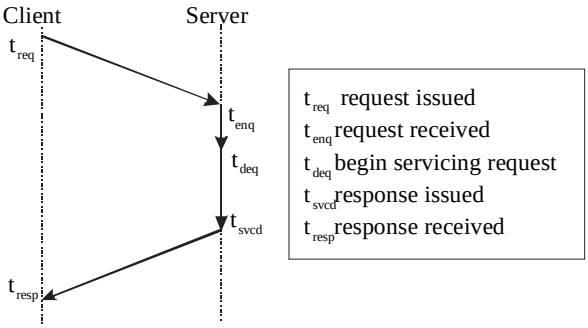


Figure 1: Timing Client/Server Interactions

A client measures the elapsed time between the issuance of a request to receipt of its response. Services measure the elapsed time from receipt of a request to issuance of its response, and include this measure in the response. Reporting server-side time allows clients to consider networking costs independently of service variations. Figure 1 illustrates these measurements. Note that only elapsed times at a single host are measured. Therefore, clocks need not be synchronized, but for server-side and end-to-end costs to be comparable, they must run at roughly similar rates.

These observations serve two purposes. First, they are used to establish a *baseline*, the performance that a client expects under normal circumstances. Second, clients use recent observations to derive a current estimate of the costs to access a service, and compare that against their baseline expectations.

Networking costs are divided into latency and bandwidth. Where available, clients may also track loss measurements. However, not all services expose loss information; services that use reliable transport mechanisms such as TCP or RPC coerce lost packets to high latency, low throughput, or both.

To derive a model of these costs, clients must take into account size of requests, size of response, and total round trip time. By using sequences of observations, weighted over time, clients can build a model of these costs with associated uncertainties. This estimation model is best provided as a middleware layer that clients of common request-response protocols can use.

Baseline metrics are established through the low-water marks of long-term observations. The intuition

behind this is that the baseline measure should characterize the best available service in order to determine what performance is acceptable. While it is unclear how often a particular client might see ideal service, one would expect the distribution to have a long tail, with a cluster of observations near the ideal. Once baselines are established, one can place bounds beyond which performance would be declared poor.

Accurately estimating network performance is an extremely difficult problem. This is particularly true for hosts that are far apart in terms of network topography, as traffic between them shows significant variability [27]. Ideally, a good estimator of network performance would be both *agile* as well as *stable*. An estimator is agile if it reacts quickly to true changes in network performance. An estimator is stable if it does not track transient changes in performance.

We are approaching the problem of agile, stable estimation from two directions. First, we implicitly account for variance by putting less faith in individual observations when recent observations have not been stable. Second, we explicitly report variance along with estimates, so that it can be taken into account by algorithm responsible for making replication decisions.

3.1. Implicitly Accounting for Variance

Our basic approach is to provide a low-pass filter on the estimations, much as TCP does in estimating round trip times [17]. The general form of such a filter is:

$$E_n = \alpha O + (1-\alpha) E_{n-1}$$

This says that the new estimate is based on the old estimate, adjusted by the current observation. The term α is the *gain*. It determines how much influence the current observation has over the new estimate.

Like TCP, our estimators report both the output of this filter along with some measure of variance. However, unlike TCP's RTT estimator, which uses a constant gain, we vary the gain to bias either towards agility or stability. The method we use to assign gain borrows from Kalman filters [10], which are more commonly used in communications and controls.

If past observations have been relatively stable, it is likely that they reflect the true state of the network between client and server. Since such observations are very accurate, one would increase the gain for the next observation accordingly. However, if past estimations have been accurate predictors of future observations, then the current observation has less to add, and the gain should be decreased. We measure stability of observation with a low-pass filter on the difference between consecutive observed values. Accuracy of prediction is similarly measured with a low-pass filter on the difference between an observation and its predicted values. The gains on each of these filters are set to 1/8. Their outputs are scaled to one another, and given equal weight in setting the overall gain.

3.2. Explicitly Exposing Variance

For some services, variable performance has an impact similar to consistently poor performance. This will be especially true for those services that interact with users. These services may wish to replicate to avoid variance, even if the average behavior is acceptable.

To account for such services, we explicitly expose the metric describing observation stability. This is similar to TCP's use of both RTT estimates and deviation in calculating retransmission timeout intervals.

4. Finding a WayStation

When a client discovers that it might benefit from replica creation, it must find a WayStation that is close enough to provide that benefit. The performance parameters provided by the estimator, combined with the expected gains from replication for a particular service, bound how far a WayStation can usefully be. We plan to have clients find WayStations within this limit by a process called *distance-based discovery*, a cost-limited multicast issued by a client in search of a replica site.

WayStations are under the administrative control of their owners, and are assumed to be managed as if they were servers for that domain's user population. WayStations will most often provide replica services to users within that population, though they can serve mobile users who are visiting that domain. Servicing such users depends on being able to authenticate them as valid visitors, and users must also establish their trust in WayStations; one could use a public-key infrastructure [14] to provide for this capability. Taken together, the WayStations form a confederation of cooperative servers.

In our model, WayStations belong to a particular multicast group, and announce their presence to routers through IGMP advertisement [7]. A distance-based multicast to this group gives latency and bandwidth limits. When a router sees such a multicast, it forwards it across links that have advertised recipients for that multicast group, but will not cause the declared limits to be exceeded. In order to support distance-based discovery, routers must estimate the costs — latency and bandwidth — to traverse the links to each of their neighbors.

Latency is an additive cost, but bandwidth is limited only by the slowest link along the path. Therefore, as packets are forwarded the expected latency is subtracted from the recorded latency limit. The recorded bandwidth limit is left unchanged, but routers record the minimum bandwidth of any traversed link. Any process that re-

ceives a cost-limited multicast is given total latency and minimum bandwidth of the traversal.

Figure 2 illustrates distance-based discovery. Boxes represent hosts; circles are routers. The lightly-shaded host sends a discovery packet with a latency limit of 30; latency estimates are given on each arc, and are additive. Each router prunes paths that are estimated to be beyond the requested bounds, and forwards along other links. The heavily-shaded boxes are the two hosts that receive this cost-limited request.

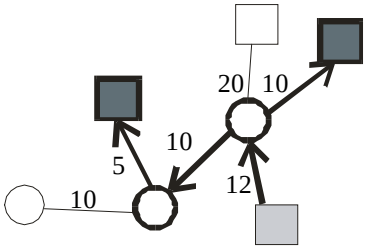


Figure 2: Distance-Based Discovery (limit 30)

Distance-based discovery has several desirable properties. First, routing decisions are made based on purely local information. Only the observations of data sent and received from neighbors are needed for a router to decide whether to prune or forward a request. Second, since each router estimates and applies costs independently, clocks between routers do not need to be synchronized. Third, messages are only routed to nodes that are within requested bounds. This is unlikely when using the time-to-live field to limit hop counts as an approximation of communication costs, since hop limits must be liberal to find all candidates. One way to avoid liberal hop limits is to use an *expanding ring search* [34], starting with low time-to-live values and increasing them to find a candidate. Unfortunately, there are potentially many wasted (albeit short) round-trips in such a scheme.

The discovery packet contains the name of the service it wishes to replicate in addition to cost bounds. A WayStation that receives this query responds with its own name, the observed cost to reach this WayStation, and whether or not that service already exists on the WayStation. The client can then select the closest WayStation, giving preference to any that have already begun to populate the requested replica, taking advantage of potentially hot caches.

Of course, networking costs to reach WayStations are only part of the picture; the load on that WayStation is also of concern. There are two ways one might deal with this. First is to allow each WayStation to benefit from cluster-based replication, isolating the problem of service load from that of network load. Second, one can incorporate service load and network costs into a single metric. However, doing so would be quite complicated.

We plan to build a prototype of this mechanism along with simple heuristics to measure the costs between routers. Layering this capability atop network technologies such as ATM that provide flow control is straightforward; one can use the flow control messages to estimate latency and provide bounds on the available throughput. Other interconnects will have to insert probe messages to provide latency bounds, and use traffic monitoring mechanisms such as Cisco's NetFlow [29] to estimate the bandwidth available along each of its links.

5. Using a Replica

After selecting a WayStation, the client asks it to create a replica of the service in question. This involves no data copying; the replica is populated lazily. However, the WayStation must inform the home service that the replica is being created. The home service uses this information to set up any state that it will need to manage replica consistency. Logically, each WayStation forms a two-way replica with the home service; the service itself manages the issues of multi-site, dynamic replication.

After establishing a replica on a WayStation, the client directs all of its read and write requests to it. If a read request arrives that cannot be satisfied, the WayStation fetches the relevant data from the remote service on demand. If there is known locality in the access pattern, the WayStation can exploit it by prefetching data where appropriate. For example, if a file system client asks for the first block of a file, the WayStation may asynchronously fetch the rest, as it is likely to be needed soon [1]. Writes are sent from the client to the WayStation as normal, but may not be reflected back to the home service immediately, depending on the consistency policy. These writes form a virtual log at the WayStation.

The key to good performance in fluid replication is choosing the appropriate consistency policy. The client created the replica in response to finding itself far from its home service. Distance-based discovery placed this replica close to the client, and therefore the replica is also very likely to be far from the home service. Substantial communication between the WayStation and remote service would be the limit on performance.

Consistency mechanisms can be described along two different dimensions: the strength of guarantees provided by the mechanism and the frequency with which those guarantees established. The latter is called the *replica maintenance interval*. Together, these two define the way in which clients perceive the consistency of objects.

As a side effect of maintaining consistency, fresh copies of data migrate from replica to replica. Schemes can be further classified by how aggressively they propagate data, and whether or not clients offer hints as to how data should be propagated. Issues of data propagation do not have any impact on the perceived consis-

tency of objects, only on the performance of using that data. Services specify a default consistency policy for the data they provide, but clients can choose to weaken or strengthen that policy based on their needs. Therefore, the system must handle conflicting consistency mechanisms within a replica set.

5.1. Strength of Guarantee

There are three different strengths of guarantee that fluid replication provides. The simplest, and least powerful, is *last-writer*. In last-writer, no effort is made to ensure that conflicts do not occur, nor are conflicting updates detected. Instead, during replica maintenance, each replica notifies the other of any stale objects. If one replica has modified a stale copy, that modification may supercede the intervening update from another site; the order in which conflicting updates are applied is undefined. Each last-writer replica maintains an update log. This is used to decide what changes need to be reflected at a replica's peer, avoiding a full replica scan during consistency maintenance. Last-writer consistency is useful for services that do not offer a strict notion of consistency, such as the Web [7] or NFS. For example, dissemination of soft-state updates [8], or software releases can be handled trivially and efficiently with this mechanism. It can also be useful in settings where updates are used primarily as hints [21].

The next strongest consistency guarantee is *optimistic* [13,18]. In optimistic consistency, no effort is made to prevent inconsistent operations. However, inconsistent updates are reliably detected and not allowed to propagate further. This guarantee is useful for common file system tasks, or any other workload where the incidence of write-sharing between users is rare.

Optimistic replica sites maintain update logs, stamped with logical clock time [20]; each cached object retains its logical last-modified time. During consistency maintenance, the WayStation sends its log to the remote service, which compares the two logs, using the logical timestamps to check for serializability. Serializable operations are applied. If an operation is judged to be non-serializable, the service checks to see if the operation can be resolved with either knowledge of the data structure or by application-provided code [19]. If this is impossible, the object is marked in conflict. Conflicting objects must be resolved by hand before they can be used.

The final consistency level is *pessimistic*. In pessimistic, or strict, consistency, all operations are guaranteed to be serializable. This guarantee is provided by requiring a replica that wishes to update an object to first acquire exclusive access to that object. This is similar to the consistency model provided by Sprite [1]. The performance benefits of pessimistic consistency in fluid replication are derived from locality in access patterns; the more locality shown by update traffic, the better pessimistic replicas will perform compared to direct use of the remote service.

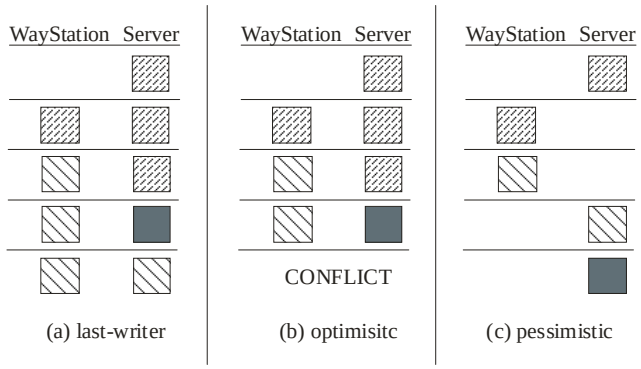


Figure 3: Comparing Consistency Guarantees

Figure 3 illustrates the difference between the three consistency guarantees. In each figure, a WayStation and a remote service see concurrent requests to modify the same object. The topmost boxes are the object in its original state. The striped and solid boxes are new versions of that object, updated at the WayStation and server respectively. In last-writer semantics, either update might take precedence over the other; the client whose write is lost will never be notified of that fact. In optimistic semantics, the conflicting update is allowed to occur, but the object is marked in conflict when update logs are exchanged. Pessimistic semantics prevents the conflicting update, but at a performance cost; the second mutator is forced to wait for the first to complete.

A central issue in providing last-writer or optimistic replication is the size of update logs. Each node must retain all log entries that mention updates that another replica has not yet seen. Such log records are necessary for update propagation in last-writer and optimistic schemes, and are also used for conflict detection for optimistic replicas. However, there are two classes of optimizations that can be made; both depend on the notion of a *replica interval*.

Let W be a WayStation that holds a replica from server S. From W's creation until its destruction, it interacts with S on k separate occasions, at times $T_1 \ldots T_k$. A WayStation's replica intervals are the k-1 periods between any two interactions; a service's replica intervals are defined as the periods between any two interactions from any WayStations holding replicas of that service.

The first class of log optimizations are the elimination of redundant or self-canceling sets of operations within a replica interval [18]. Redundant operations include updates to the same object; only the last update need be maintained. Self-canceling operations are similar. For example, suppose a file is created and then deleted within a replica interval; from any other observer's point of view, it is as if those updates never occurred.

The second class of optimization is truncating a log's *stable prefix*; removing the log entries that are known to be unnecessary from now on. For a WayStation, the stable prefix of the log is the portion before the last interaction with the remote service. For the service, the stable prefix is defined as the portion of the log known to all replicas. It is the latest point of the log that is before the most recent interactions from all replicas. Because all replicas are known to the service, it is easy to determine the stable prefix.

5.2. Frequency of Guarantee

Pessimistic consistency must be performed aggressively, prior to each update. However, optimistic and last-writer schemes can vary the frequency with which they exchange updates. This frequency is controlled in concert by the WayStation and remote service. Since each WayStation interacts only with the remote service, we can provide bounds on update propagation by restricting the interval appropriately.

There are two considerations in selecting an exchange interval. As update rates at the WayStation or the remote service increase, updates should be exchanged more frequently to reduce the chance of seeing stale data or producing inconsistent updates. However, as the network path between replica sites degrades, one might wish to defer exchanging updates to benefit from their locality [1,18]. The degenerate case — infinite time to exchange — can be used for data known to be read-only, or data for which updates are not shared, such as mirror sites. How to best balance these concerns is an open question, and one we are actively exploring.

The chief pitfall in selecting a replica maintenance interval is scalability. Because all WayStations are replicas of the remote service, that service must see all of their updates. However, without fluid replication, each service would be dealing with clients directly. Since multiple clients may use the same WayStation for updates, and optimizations can be applied per-WayStation, this may actually improve the scalability of the remote service. Services can also use cluster-based techniques to further enhance scalability. Finally, one can imagine a hierarchy of replica sites if scalability is of chief concern, but doing so while providing update bounds requires careful thought.

5.3. Update Propagation

When a replica site discovers that its peer has updated an object that it stores, it can either invalidate its copy of the object, or it can aggressively retrieve the object from its peer. The best alternative depends on a number of factors: the locality of updates, the degree and frequency of sharing, and the performance of the network path between replicas. The replication system can monitor these, and pick the option that best fits current access patterns. When applications have some special knowledge of data access patterns, they can offer hints by tagging data with expected migration patterns, similar to the annotations offered by Munin [4]. As with selection of maintenance interval, the decision of whether to invalidate or propagate depends on information from the remote service and the WayStation. This allows the propagation of data to be done strictly for performance reasons, and need not impact the performance along the critical path of any clients.

5.4. Handling Mechanism Conflicts

The default consistency scheme for data is chosen by the home service based on service semantics and expected data access pattern. However, clients that use WayStation replicas can ask for different consistency mechanisms when appropriate. Each set of WayStations with the same strength of guarantee forms a *consistency class*. Thus, each replica set can have three replica classes: last-writer, optimistic, and pessimistic semantics. This allows WayStations to degrade their class when stronger guarantees are too expensive to provide. It also allows a client to provide *session semantics* [35] when changing from one WayStation to another; this is described in more detail in Section 6.

When conflicts arise between replica classes, the stronger guarantee prevails. For example, a replica with a last-writer class and a pessimistic class will always guarantee that the pessimistic class' updates supercede those of the last-writer class. This preserves stronger guarantees by placing the burden of dealing with inconsistency with weaker classes, where it is already deemed acceptable.

One question is whether or not each class must further divide itself; for example, should isolated groups of WayStations use strict consistency within them, but weaker consistency between them. This can only be answered through experimentation and use. The range of possible inter-class and intra-class conflicts, and the actions taken when they arise, are summarized in Table 2.

conflict btwn	last-writer	optimistic	pessimistic
last-writer	either persists, no guarantee	optimistic persists, no conflict	pessimistic persists
optimistic		conflict	pessimistic persists, optimistic conflicts
pessimistic			updates serialized

Table 2: Handling Mechanism Conflicts

6. Destroying and Migrating Replicas

There are two reasons why a WayStation replica might be destroyed. First, the client may cease to be interested in that replica's data. Second, the client might move closer to another WayStation or the home service.

The former case is easy to deal with. WayStations can monitor the usage statistics of their replicas. Those that have not been used recently can be marked *dormant* as soon as their changes have been reflected to the central service. A busy WayStation can reclaim the resources of dormant replicas when necessary.

Handling a moving client is more difficult. When a client moves, it must see *client-consistent* updates; if a client performs an update, it should be persistent from the client's point of view, according to the consistency rules in effect. This problem is illustrated in Figure 4.

The client first updates an object on its WayStation, and then moves to a second WayStation. If the first replica is using optimistic or last-writer semantics, the client may not see its own prior update, since the departure WayStation has not yet propagated it to the service.

Individual clients expect client-consistent updates. Put another way, no one should know more about a client's updates than that client does. In the Bayou terminology, this is known as *read-your-writes* session semantics [35]. Pessimistic consistency schemes provide client-consistency automatically; more relaxed schemes may not. To provide client-consistency, a client must ask its WayStation to discontinue replication on its behalf on departure. Conceptually, the departure WayStation must then propagate all uncommitted updates to the remote service before allowing the client to begin using a new replica site. Since this can be quite expensive, there are several optimizations that one might make.

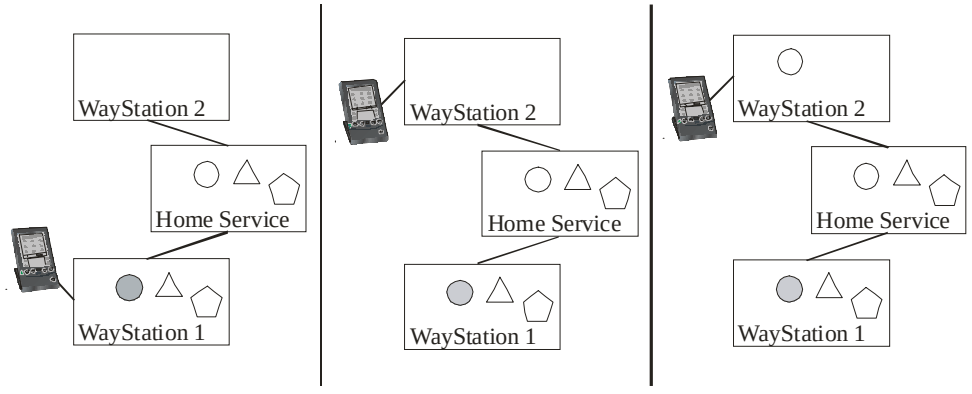
The first optimization depends on the fact that the client itself may have cached some of its most recent updates. To capture this notion, the client maintains a *log suffix*, the log timestamp such that the contents of all updates stamped with later times are known to the client. As updated data are evicted from the cache, the log suffix pointer moves forward through the log.

When notifying a WayStation of its departure, the client sends its current log suffix. The departure WayStation is then responsible for immediately propagating pre-suffix updates. It can purge post-suffix updates, and have the client replay them at the arrival WayStation after being given its release. Since the arrival WayStation was chosen based on its proximity to the client, this replay operation will be fast.

The second optimization is based on the observation that the arrival WayStation is likely to be closer to the departure WayStation than the remote service; this will be true if the client has not moved far in terms of network topography. In these cases, the departure WayStation is free to send its update log and file contents to the arrival WayStation. This defers the expense of committing changes to the remote service to some time after the client changes replica sites.

Neither of these optimizations is guaranteed to improve performance in all situations; the actual gains depend on relative communications costs. However, given the mechanisms to establish replicas and choose WayStations, they hold promise. The main concern is the effect on consistency of these optimizations; they defer previously issued updates to a later logical time. There are issues with doing so correctly that we must address.

The final optimization makes use of the consistency class mechanism to defer even more work from the critical path of replica handoff. Rather than propagate changes, the departure WayStation can promote the consistency scheme to pessimistic, and invalidate modified replicas at the home service rather than force an update.



(a) update on WS1 (b) move to WS2 (c) fetch stale object

Figure 4: Violating Client-Consistency

This exchange of status information is fast, and will allow data propagation to be overlapped with other client operations. Note that the use of consistency classes allows this change to affect only the performance of the arrival and departure WayStations without penalizing replica sites that choose weaker consistency guarantees. Also, this optimization, unlike the first two, changes neither the currency of updates nor their position in logical time.

7. Current Status

This work is in its infancy; we are only beginning to implement and experiment with these ideas. Our current focus has been on network estimation filters and the construction of a simple prototype to explore the performance space.

The client's network estimation service forms the core of fluid replication; without good estimators, the system will adapt sluggishly, capriciously, or both. While we have only tentative results for the quality of our estimator, they are promising. We are able to track changes in bandwidth and latency with good agility, but our estimators do suffer from some instability. The exposure of estimate error alleviates this, and we believe that incorporating it into the decision algorithm will allow us to make much better decisions about replica instantiation.

Our prototype consists of a WayStation, client, and server to add fluid replication to NFS. We are incorporating the two extremes of consistency: last-writer with an infinite write-back interval, and immediate write-through. When completed, it will allow us to better understand the gains possible for replication of this service, and help us tune the decision algorithm.

8. Related Work

The idea of using replication to improve the performance of wide-area systems is certainly not new. Grapevine [31] was the first distributed system that used replication with weak consistency to provide good performance and scalability. It also served as the genesis of fluid replication. The observation that replicas should be placed at either end of a slow link led us to wonder how to position replicas automatically.

Cluster-based, scalable distributed services focus on replication within a tightly coupled cluster to adapt to changing client load. Typically, they focus on soft-state replicas [8] or provide back-end storage that is shared across the cluster [26]. This minimizes the overhead of consistency maintenance, though more recent systems have explored dynamic content [6]. These systems provide good scalability in cases where the end-service is the performance bottleneck, but do not deal with other sources of performance problems.

Several distributed systems have used an optimistic consistency scheme to provide file and database access to mobile clients; such clients experience significant variation in the networking costs incurred in accessing distributed services. Ficus [13] and Bayou [36] rely on a peer-to-peer replication model. In this model, each node stores a replica, and pairs of nodes exchange updates when they encounter one another. This provides eventual consistency, but cannot bound time to convergence. JetFile [12] also provides a peer-to-peer model, but allows peers to find each other by IP multicasts that are global rather than directed only to nearby neighbors.

Bayou's use of update logs and replica management [28] is in many ways similar to that proposed here. However, in Bayou these mechanisms must all work with arbitrary replication topologies. Restricting their use to pair-wise interactions between WayStations and servers is likely to simplify them significantly in addition to providing the foundation for bounded convergence.

Coda [18,23] provides a cluster-based replication model, but allows clients to hold long-term, *second-class* replicas. These replicas, stored solely on clients, are subject to the decreased safety and security offered by them. Coda also has mechanisms to deal with *weak connectivity* [23], but they require potentially expensive interactions with remote services.

Web caches take advantage of locality in HTTP requests to provide better performance to clients using them [22]. These caches are limited to the consistency mechanisms provided by the Web, and are passive; they do not accept updates from clients. We know of no systems that allow clients to transparently migrate between caches as the costs of communication change.

WebOS [37], a set of abstractions for building wide-area applications, is complimentary to our own work in many ways. WebOS' naming, security, and process control mechanisms could serve as the foundation for implementing fluid replication. In fact, one of the applications chosen to demonstrate the benefit of the WebOS abstractions is a simpler form of fluid replication called *Rent-A-Server*. WebOS also provides a wide area file system that supports last-writer semantics for general file access, along with stronger consistency for server-push and append-only data. While WebOS concentrates on the building blocks one might use to create an automatic replica management system, our concern is that system itself.

Distributed shared memory systems take advantage of locking mechanisms to optimize data movement and invalidation [2,11,16]. Programs that correctly lock data see pessimistic consistency semantics. Munin [4] enables cooperative applications to annotate data to

expose application knowledge to further optimize data movement. Khazana [5] is a distributed middleware service that provides an abstraction similar to distributed shared memory; a flat, byte-addressable space shared by all participating nodes. It is structured to allow easy experimentation with consistency protocols, reflecting each lock and update request on behalf of a client to a local consistency manager. To date, Khazana has explored only pessimistic consistency, though the consistency manager architecture it uses [3] could serve as a useful testbed to explore schemes with weaker guarantees.

GeoCasting [24] provides broadcasts that are limited to a physical area. Distance-based discovery combines this idea with schemes that estimate point-to-point network costs to broadcast based on network distance rather than physical location. There are several systems that provide network performance estimation services, including IDMaps [9], the Network Weather Service [38], and SPAND [32]. However, they tend to be geared towards problems such as the *mirror selection problem*: how to chose among a known set of servers with which the client has had little recent interaction. Our problems differs from this one in two ways. Clients that must estimate performance to a server they are already using have substantial knowledge on which to base a decision. When that client must select a WayStation, it may have no idea where candidates might be found; this is particularly true for mobile clients.

9. Conclusion

Fluid replication promises to address the sources of many performance problems in large scale, wide-area, distributed systems. It reacts to changes in demand for services and resources by automatically replicating those services when and where necessary. These replicas are placed on WayStations — service nodes in the infrastructure that provide replication services.

There are several areas in which fluid replication will make novel contributions. It provides local mechanisms to dynamically estimate and react to changing network conditions. It explores techniques for local resource discovery and provisioning, and implements and compares a rich set of coexisting consistency mechanisms. Finally it provides a framework to adaptively set parameters for consistency management.

In addition to providing improved performance for distributed services, fluid replication will be a useful tool in reducing administrative costs for large installations. By automatically provisioning services based on client demand and resource availability, we can best take advantage of prevailing conditions for distribution and sharing of data.

Acknowledgements

We wish to thank David Barkovic, Mark Corner, and Jim Zajkowski, who helped us refine both the ideas in and presentation of this paper. We also wish to thank our anonymous reviewers for their helpful suggestions and improvements. This work was supported in part by IBM and Novell. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of IBM, Novell, the University of Michigan, or the State of Michigan.

Bibliography

1. Baker, M. G., J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, p. 198-212. Oct 1991, Pacific Grove, CA, USA.
2. Bershad, B. N., M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. *COMPCON Spring '93. Digest of Papers*, p. 528-37. Feb 1993, San Francisco, CA, USA.
3. Brun-Cottan, G., and M. Makpangou. 1995. "Adaptable replicated objects in distributed environments." *Rapport de Recherche INRIA*, No. RR 2593.
4. Carter, J. B., J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. *Thirteenth ACM Symposium on Operating Systems Principles*, p. 152-64. Oct 1991, Pacific Grove, CA, USA.
5. Carter, J., A. Ranganathan, and S. Susarla. Khazana: an infrastructure for building distributed services. *Proceedings of 18th International Conference on Distributed Computing Systems*, p. 562-71. May 1998, Amsterdam, Netherlands.
6. Challenger, J., A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. *Proceedings of IEEE INFOCOM '99*, p. 294-303. Mar 1999, New York, NY, USA.
7. Fenner, W. 1997. "Internet group management protocol, version 2." *Internet RFC 2236*.
8. Fox, A., S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, p. 78-91. Oct 1997, Saint Malo, France.
9. Francis, P., S. Jamin, V. Paxson, L. Zhang, D. Gryniewicz, and Y. Jin. An architecture for a global Internet host distance estimation service. *Proceedings, IEEE INFOCOM '99*, p. 210-17. Mar 1999, New York, NY, USA.
10. Gelb, A., Editor. 1974. *Applied Optimal Estimation*. Cambridge, MA, USA: M.I.T. Press.
11. Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *The 17th Annual International Symposium on*

Computer Architecture, p. 15-26. May 1990, Seattle, WA, USA.

12. Gronvall, B., A. Westernlund, and S. Pink. The design of a multicast-based distributed file system. *Proceedings of the Third Symposium on Operating Sytsems Design and Implementation*, p. 251-64. Feb 1999, New Orleans, LA, USA.
13. Heidemann, J. S., T. W. Page, R. G. Guy, G. J. Popek, J.-F. Paris, and H. Garcia-Molina. Primarily disconnected operation: experience with Ficus. *Proceedings of the Second Workshop on the Management of Replicated Data*, p. 2-5. Nov 1992.
14. Housley, R., W. Ford, W. Polk, and D. Solo. 1999. "Internet X.509 public key infrastructure certificate and CRL profile." *Internet RFC 2459*.
15. Howard, J. H., M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems* 6, no. 1: p. 51-81. Feb.1988
16. Iftode, L., J. P. Singh, and K. Li. Scope consistency: a bridge between release consistency and entry consistency. *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, p.451-73. Jun 1996, Padua, Italy.
17. Jacobson, V. Congestion avoidance and control. *SIGCOMM '88 Symposium: Communications Architectures and Protocols.*, p. 314-29. Aug 1988, Stanford, CA, USA.
18. Kistler, J. J., and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Transactions on Computer Systems* 10, no. 1: p. 3-25. Feb.1992
19. Kumar, P., and M. Satyanarayanan. Flexible and safe resolution of file conflicts. *Proceedings of the 1995 USENIX Technical Conference*, p. 95-106. Jan 1995, New Orleans, LA, USA.
20. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, no. 7: p. 558-65. July1978
21. Lampson, B W. Hints for computer system design. *IEEE Software* 1, no. 1: p. 11-28. Jan.1984
22. Luotonen, A., and K. Altis. World-Wide Web proxies. *Computer Networks and ISDN Systems* 27, no. 2: p. 147-54. Nov.1994
23. Mummert, L. B., M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, p. 143-55. Dec 1995, Copper Mountain Resort, CO, USA.
24. Navas, J. C., and T. Imielinski. GeoCast -- geographic addressing and routing. *Proceedings of Third ACM/IEEE International Conference on Mobile Computing and Networking 1997 (MobiCom'97)*, p. 66-76. Sep 1997, Budapest, Hungary.
25. Noble, B. D., M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. In *ACM SIGCOMM '97 Conference. Applications, Technologies, Architectures, and Protocols for Computer Communication*, p. 51-62. Sep 1997, Cannes, France.
26. Pai, V. S., M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. *ASPLOS-VIII. Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* , p. 205-16. Oct 1998, San Jose, CA, USA.
27. Paxson, V. End-to-end Internet packet dynamics. *ACM SIGCOMM 97 Conference. Applications, Technologies, Architectures, and Protocols for Computer Communication*, p. 139-52. Sep 1997, Cannes, France.
28. Petersen, K., M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, p. 288-301. Oct 1997, Saint Malo, France.
29. Presti, K. Cisco enhances IOS NetFlow router software to aid ISPs. *Computer Reseller News*, no. 745: p. 106. July1997
30. Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. *USENIX Association Summer Conference Proceedings 1985*, p. 119-30. Jun 1985, Portland, OR, USA.
31. Schroeder, M. D., A. D. Birrell, and R. M. Needham. Experience with Grapevine: the growth of a distributed system. *ACM Transactions on Computer Systems* 2, no. 1: p. 2-23. Feb.1984
32. Seshan, S, M Stemm, and R H Katz. 1997. SPAND: shared passive network performance discovery. in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, p.135-46. Dec 1997, Monterey, CA, USA.
33. Shneiderman, B. 1998. *Designing the user interface: strategies for effective human-computer interaction.* 3rd edition ed. Reading, MA, USA: Addison-Wesley.
34. Stevens, W. R. IGMP: Internet group management protocol. *TCP/IP Illustrated.*, p. 179-86. Vol. 1. Reading, MA, USA: Addison-Wesley Publishing Company.
35. Terry, D. B., A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, p. 140-9. Sep 1994, Austin, TX, USA.
36. Terry, D. B., M. M. Theimer, K. Petersen, and A. J. Demers. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, p. 172-83. Dec 1995, Copper Mountain, CO, USA.
37. Vahdat, A., M. Dahlin, T. Anderson, E. Belani, D. Culler, P. Eastham, and C. Yshikawa. WebOS: operating system services for wide area applications. *Proceedings The Seventh International Symposium on High Performance Distributed Computing*, p. 52-63. Jul 1998, Chicago, IL, USA.
38. Wolski, R., N. T. Spring, and J. Hayes. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*. to appear