Fluid Replication

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

Abstract

The performance of distributed services is becoming increasingly variable due to changing load patterns and mobile users. Current approaches — cluster-based, scalable services and peer replication — solve only part of the problem. The former applies only to end-service variability, while the latter compromises safety and the ability to offer bounds on consistency of updates. We propose *fluid replication*, the ability to create a replica where and when it is most 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

The performance of distributed services is becoming increasingly variable. There are several reasons for this trend. Over time, aggregate user interests tend to change rapidly and unpredictably — a phenomenon known as the *slashdot effect* — placing variable demands on end-service and networking resources. Further, changing patterns of demand in the underlying network also lead to variable performance, even if the demand for the service itself has not changed. Finally, users are becoming increasingly mobile; as they move, the cost of accessing their home services change.

Manual administration of this flexible world is clearly not up to the task. There are useful heuristics; for example, one might place replicas on either side of a slow network link. However, manual intervention is too cumbersome to keep pace with the rapid changes in demand for services. Furthermore, resources are often dedicated to trouble spots that do not persist; as load decreases, those resources are not reclaimed.

Cluster-based, scalable services have been used to address some of the sources of performance variability. In this approach, services are deployed on a tightly coupled cluster of machines. When one service discovers heavy load, it replicates itself elsewhere on the cluster; this replica is reclaimed when it is no longer needed. This technique is very effective when the end service is the bottleneck, but does not address other sources of variability.

In contrast, autonomous, peer-replicated systems place the responsibility for replication with the clients. In this approach, each client caches a subset of the data in which it is interested. It can then operate on that data locally, exchanging its updates with another client when they encounter one another. This addresses the question of network-induced variations in performance, but has its own difficulties. Because update propagation depends on user mobility patterns, one cannot offer bounds on update convergence or consistency. When a client overruns its local caching resources — a possibility on extremely lightweight, low power devices — it must fall back to its original, remote service. Finally, updates that are stored only on clients are more vulnerable 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 convenience of client-based schemes. We propose *fluid replication* — the easy creation of replicas where and when they are most needed — as a way to provide these properties. In fluid replication, clients and services monitor their observed performance; when it becomes poor, a replica is created to either handle the increased load or avoid the poor network path. Central to our approach is a new abstraction, the WayStation. WayStations are service nodes throughout the infrastructure that provide remote resources on which to place replicas.

There are several challenges that must be met to realize this vision. First, we must have a way to decide that we need a new replica, and a mechanism to select a WayStation on which to host it. Once we have established a replica, we choose consistency policies to balance the consistency and performance seen when using that replica. Finally, we need a way to reclaim WayStation resources in a way that preserves the consistency properties that clients expect.

2. Replica Creation

The fundamental abstraction in fluid replication is the WayStation. WayStations are nodes within the infrastructure that provide resources to users. These nodes are managed by their local administrative domain; there is no notion of centralized administration. A WayStation can potentially provide resources to any user, including those from other domains.

Each client and server monitors the performance of requests and responses. Clients measure end-to-end response time of their requests; servers measure the time requests spend under their control. The difference between these two measures the communication costs involved in servicing the request.

If the server discovers that service time has grown unacceptably, it creates a *local replica*. A local replica is located close to the service, on the same cluster of machines. Client requests can be redirected by a front-end service to provide load balancing, and the high-bandwidth of the cluster can support strong consistency. If consistency becomes expensive, data can be repartitioned to avoid sharing across servers [4,6,18].

If the client discovers long communication times, it will initiate a replica on a WayStation that is close to the client — placed so that communication with it is inexpensive — and able to absorb the added workload. We use a mechanism called *distance-based discovery* to identify candidate WayStations for this replica. Distance-based discovery is a form of local, limited broadcast where the notion of locality is defined by network costs.

In order to support distance-based discovery, routers must estimate the costs — latency and bandwidth — to traverse each of their links. A distance-based broadcast gives latency and bandwidth limits. When a router sees such a broadcast, it forwards it only across links that will not cause the declared limits to be exceeded. The operating system delivers the payload and the cost incurred to any receiving process.

To locate a nearby WayStation, a client sends a cost-limited broadcast containing the name of the service it wishes to replicate. A WayStation that receives this query responds with its own name, the observed cost to reach this WayStation, whether or not that service already exists on the WayStation, and an estimate of its current load. The client can then select the closest, lightly-loaded WayStation, giving preference to any that have already begun to populate the requested replica.

The client then asks the chosen WayStation to create a replica of the service in question. This involves no data copying; the replica is populated lazily. However, in most cases 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.

The home service specifies a default consistency policy, but the client can ask for a different one if circumstances warrant. Logically, each WayStation treats only to the central service as its replica peer; the central service is responsible for coordinating between WayStations. This helps manage the complexity of a replication system where replicas come and go at will. All WayStations using the same consistency model are gathered together into a replica class. This means that clients who ask for strict consistency will see each others' updates, but may not see updates from clients that have asked for more relaxed consistency.

3. Replica Operation

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 as well 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 to the home service immediately, depending on the consistency policy. These writes form a virtual log, though not all consistency require that the log be stored explicitly.

Choosing the appropriate consistency policy is critical to good client performance. This is because the client created the replica in response to finding itself far from its home service. This replica was placed close to the client, and therefore is also very likely far from the home service. Substantial communication between the WayStation and remote service will likely be the limit on performance.

Consistency schemes can be described along two different dimensions: what notion of consistency is provided and how often consistency is maintained. 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.

There are three different levels of consistency we plan to provide. The simplest, and least strict, is *last-writer* consistency. In last-writer consistency, no effort is made to ensure that conflicts do not occur, nor are conflicts detected. Instead, during replica maintenance, each replica notifies the other of any stale objects. If one replica has a stale copy that is modified, that modification may supercede the intervening update at another site; the order in which two replicas' updates are applied is undefined.

Each last-writer replica maintains an update log, but uses it only to decide what changes need to be reflected at its peer, to avoid a full replica scan. Last-writer consistency is useful for services that themselves do not offer a strict notion of consistency, such as the Web[5] or NFS [11].

The next consistency level is *optimistic* [10,13]. In optimistic consistency, no effort is made to prevent inconsistent operations, but inconsistencies are detected and not allowed to propagate further. When possible, inconsistencies are resolved automatically by the system or application-specific code [14].

Optimistic replica sites keep operation logs, stamped with logical clock time. During consistency maintenance, the WayStation sends its log to the remote service, which compares the two logs checking for serializability. Operations which can be serialized are applied. When an operation is judged to be non-serializable, the service checks to see if the operation can be resolved with either system knowledge of the data structure or application-provided code.

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 provided by Sprite [1]. The performance benefits of pessimistic consistency are derived from locality in access patterns; the more locality shown by update traffic, the better pessimistic WayStations will perform compared to direct use of the remote service.

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. There are two considerations in selecting an interval. As update rates 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 replica maintenance to benefit from locality of updates [1,13]. How to best balance these concerns is an open question. 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.

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 *backfetch* the object from its peer. The best alternative depends on a number of factors; the locality of update, the degree and frequency of sharing, and the performance of the network path between replicas. The replication system can monitor some of 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, similar to the annotations offered by the Munin distributed shared memory system [3].

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. Thus, each replica set can have three replica classes: replicas with last-writer semantics, optimistic replicas, and pessimistic replicas. This allows WayStations to degrade their class when stronger semantics is too expensive to provide. It also allows a client to provide *session semantics* [20] when changing from one WayStation to another. We elaborate on this notion in Section 4.

When conflicts arise between replica classes, the stronger class always wins. 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.

4. Replica Destruction

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

The former case is easy to deal with. WayStations can monitor the usage statistics of their replicas. Those that have not been used 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.

The latter case is more difficult. When a client moves, it must see *client-consistent* updates. If a client performs an update, that update should be persistent from the point of view of the client. Another way of saying this is that no one should know more about a client's updates than the client does. In the Bayou terminology, this is known as *read-your-writes* session semantics[20]. Pessimistic consistency schemes provide client-consistency automatically; more relaxed schemes may not.

When leaving a WayStation for another replica site, the client asks the departure WayStation to discontinue replication on its behalf. 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. However, since this can be quite expensive, there are several optimizations that can be made.

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 all updates stamped with later times are known to the client. When notifying a WayStation of its departure, the client sends its current log suffix. The departure WayStation then is only responsible for immediately propagating pre-suffix updates. It can purge post-suffix updates, and the client can replay those updates 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 may be closer to the departure WayStation than the remote service. In such 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.

The final optimization makes use of the consistency class mechanism to defer even more work from the critical path of replica handoff. Rather than actually 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. 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 arrival and departure WayStations without penalizing replica sites that choose weaker consistency guarantees.

5. Related Work

Grapevine [19] was the first distributed system that provided replication with weak consistency to provide good performance and scalability. The observation that replicas should be placed at either end of a slow link led us to ask the question how one might do so automatically.

Cluster-based, scalable distributed services [6,18] focus on replicating data and services within a tightly coupled cluster to adapt to changing client load. Typically, they focus on soft-state replicas or provide a back-end storage shared across the cluster. This minimizes the overhead of consistency maintenance, though more recent systems have cached dynamic content [4]. They provide good scalability in cases where the end-service is the performance bottleneck.

Several distributed systems have used an optimistic consistency scheme to provide file and database access to mobile clients. Ficus [10] and Bayou [21] 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 does not guarantee the rate of convergence. JetFile [9] provides a similar peer-to-peer model, but allows peers to find each other by IP multicast; each multicast is global rather than directed only to nearby neighbors. In contrast to these peer-to-peer systems, Coda [13,16] provides a service-based replication model, but allow clients to hold *second-class* replicas. All of these systems store updates only on clients; these are subject to decreased safety and security compared to their server-stored counterparts.

Web caches take advantage of locality in HTTP requests to provide better performance to clients using them [15]. These caches are limited to the consistency mechanisms provided by the Web, and are passive elements; they do not accept updates from clients.

Distributed shared memory systems take advantage of locking mechanisms to optimize data movement and invalidation [2,8,12]. Programs that correctly lock data see pessimistic consistency semantics. Munin allows applications to annotate data to expose application knowledge to further optimize data movement [3].

GeoCasting [17] provides broadcasts that are limited to a physical area. Distance-based discovery combines this idea with schemes that estimate point-to-point network costs [7] to broadcast based on network distance rather than physical distance.

6. Conclusion

The cost of accessing distributed services is becoming increasingly variable through changing popularity, use of the underlying network, and client mobility. Cluster-based services solve this problem when the end-service is the source of variation, but are of limited use in other situations. Peer-to-peer replication systems can also deal with variation, but at a cost of lowered safety and unknown bounds on replica convergence.

We propose to solve the problem of service variability through fluid replication; the ability to create replicas when and where they are most useful. Well-placed replica sites are found through distance-based discovery. By taking into account service and client semantics, we can adjust consistency schemes to achieve substantial performance benefits.

In addition to consistency mechanisms and policies, there are several other questions to address in the context of this work. How can users put their faith in WayStations administered in a foreign domain? How can WayStations discriminate between clients to which they do and do not wish to provide service? How can failure of WayStations and communication paths be integrated into our model?

Our effort is just beginning. We are currently experimenting with a prototype built on NFS with a few, simple consistency schemes to evaluate the potential benefits of fluid replication. We then plan to apply the lessons we learn there to the construction of a more complete system.

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. 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 .
4. Challenger, J., A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. *Proceedings of IEEE INFOCOM'99*Mar 1999, New York, NY, USA .
5. Fielding, R., J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. 1997. "Hypertext transfer protocol - HTTP/1.1." *Internet RFC 2068.*
6. 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 .
7. 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*Mar 1999, New York, NY, USA .
8. 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 .
9. 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 .
10. 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.
11. 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
12. 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 .
13. 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
14. 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 .
15. Luotonen, A., and K. Altis. World-Wide Web proxies. *Computer Networks and ISDN Systems* 27, no. 2: p. 147-54. Nov.1994
16. 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 .
17. 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 .
18. 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. 3-7 Oct. 1998 .
19. 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
20. 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 .
21. Terry, D. B., Theimer M. M., 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 .