Open access • Journal Article • DOI:10.1023/B:MONE.0000034707.26695.E8

Roam: A Scalable Replication System for Mobility — Source link

David Ratner, Peter Reiher, Gerald J. Popek

Institutions: University of California, Los Angeles

Published on: 01 Oct 2004 - Mobile Networks and Applications (Kluwer Academic Publishers)

Topics: Optimistic replication, Mobile search and Replication (computing)

ROAM: A Scalable Replication System for Mobility[1]

David Ratner   Peter Reiher   Gerald J. Popek

Computer Science Department,
University of California, Los Angeles

Abstract

Nomadic users require replication to store copies of critical data on their mobile machines while disconnected or poorly connected. Existing replication services do not provide all classes of mobile users with the capabilities they require, which include: the ability for direct synchronization between any two replicas, support for large numbers of replicas, and detailed control over what files reside on their local (mobile) replica. Mobile users must adapt their behavior to match the level of service provided by today's replication systems, thereby hindering mobility and costing additional time, money, and systems management.

Roam is a replication system designed to satisfy the requirements of the mobile user. Roam is based on the *Ward Model*, a replication architecture for mobile environments. Using the Ward Model and new distributed algorithms, Roam provides a scalable replication solution for the mobile user. We describe the motivation, design, and implementation of Roam and report its performance.

1. Introduction

Replication provides both improved performance and reliability for mobile computers by creating multiple replicas of important data; optimism allows the replicas to be independently updated, and resolves concurrent updates when they are detected. Mobile users require optimistic replication to access shared data in the face of variable connectivity, bandwidth, and latency.

Existing replication services are designed primarily to handle disconnections, rather than mobility. While they maintain correct behavior under disconnections and network partitions, they are less helpful in assisting mobile users who find themselves at new locations.

Systems based on the *client-server* model, such as Coda [12], allow only direct update propagation between clients and servers. A client traveling across the country is forced to synchronize data via a long-distance phone call to the home server. The user must essentially pretend that mobility has not occurred, costing both time and money.

Systems based on the traditional *peer-to-peer* model, such as Ficus [5] and Bayou [14], allow peer communications, but they typically scale poorly in the number of replicas. Since mobility increases replication factors, the restriction on scale is an equal hindrance to mobility.

[1] This work was sponsored by the Defense Advanced Research Projects Agency under contract DABT63-94-C-0080. The authors can be reached at the Computer Science Department, UCLA, Los Angeles, CA 90095; by email to {ratner,reiher,popek}@cs.ucla.edu.

Roam is a new replication solution designed for mobile computing at high scales. Roam uses optimistic replication. As noted in previous research [5, 14], optimistic replication can lead to concurrent updates, but methods for reliably detecting such updates and automatically reconciling many of them have been successful. Further, practical experience shows that concurrent updates are rare [10].

Roam maintains a peer model between all replicas, allowing any two replicas to directly synchronize, while still scaling well. Roam provides an "anytime, anywhere" mobile architecture that allows users to be mobile without any special burdens.

Roam achieves scalability by using a hybrid reconciliation model and dynamic version-vector management. The hybrid *Ward Model* [8] incorporates elements of both client-server and peer solutions to handle replica management and update distribution. New dynamic version-vector management techniques address the scalability of the versioning information.

2. Motivation

A replication system for mobile use must provide four basic qualities:

1. Support for any-to-any communication
2. Support for potentially hundreds of read-write replicas
3. A file-granularity replication-control interface
4. A "get-up and go" model requiring no pre-motion actions

By definition, mobile users change their geographic location. Since it is typically cheaper, faster, and more efficient to communicate with a local partner than a remote one, mobile users want to communicate and synchronize with nearby replicas. This requirement implies a peer replication model.

Peer models face several challenges, but a particularly large one is scalability. Most replication systems are designed for scenarios that require only small numbers (fewer than a dozen) of writable replicas. However, mobile environments of the future may require many more replicas.

Each mobile user requires a replica of shared data on his own machine, possibly increasing replication factors for some files into the low tens. In the near future, instead of each user having one stationary and one mobile machine, there will be many more "smart" devices capable of storing replicated data, such as palmtops, PDAs, and even watches. This trend could increase the number of replicas required by more than an order of magnitude. For maximum flexibility and functionality, we must provide the ability for all replicas to generate updates, although some may never do so.

Gray's [1] critique of the scalability of peer replication is based on transactional databases. Some assumptions underlying his argument are incorrect for file systems, and thus his results do not apply to this case.

Many systems requiring one item in the replication container must store the entire container. Mobile computers have much less disk storage available than desktop machines, especially desktops using remote file servers. A mobile replication system should replicate at a granularity that avoids storing unnecessary files [9, 7].

Mobile users prefer not to prearrange their mobility, so it is unreasonable to require them to perform registration before moving. Further, mobility cannot always be predicted or scheduled. We cannot require that users know *a priori* either when they will become mobile or for how long.

3. The Ward Model

Combining client-server and peer replication architectures can better meet the requirements of some classes of mobility. Roam combines a peer architecture's ability for replicas to communicate with its client-server model features, forming a hybrid architecture called the Ward Model [8]. We cluster replicas into peer-based groups to obtain good scalability, allowing simple, cheap communications within the group. We allow group membership to change dynamically without global coordination, providing any-to-any communication capabilities between all system participants. Our group-changing operations provide a "pay as you go" framework, so the cost of mobility is roughly proportional to its duration.

3.1 Wards

We group replicas into *wards* (wide area replication domains). Wards capture the notion of typical communication partners. For example, given four replicas in Los Angeles and four replicas in New York, we might build one ward in each city. Replicas would usually pay local costs to propagate updates to each other. All ward members are peers, allowing any pair of ward members to directly synchronize and communicate.

Wards are a collection of "nearby" replicas. Parameters like geographic location, bandwidth, latency, cost, and sharing patterns would ideally be considered in forming wards.

We replicate using the *volume* data structure. A volume is smaller than a file system but larger than a directory [13]. For maximum flexibility, each shared volume has its own ward structure.

Each ward has a *ward master* that maintains consistency with the other wards. The ward master has some similarities to a server in a client-server system, but there are several important distinctions.

First, any ward member can be the ward master, since all ward members are peers. When a ward master fails, any other ward member can be elected to replace it. In client-server models, the clients cannot become servers, and no data can be propagated if the server fails.

Second, the ward master need not physically store all data objects. The ward master only needs to be able to name the intra-ward objects. In a client-server model, the server must store a complete super-set of all objects.

Third, the ward master is not a bottleneck for intra-ward synchronization. Any ward member can directly synchronize with any other ward member.

Finally, the ward members and the ward master are distinguished only in name and additional responsibilities, rather than actual control mechanisms. Ward masters and other ward members differ by less than fifty lines of C++ code.

The ward master synchronizes all intra-ward data with other wards. The ward master effectively belongs to two wards: the local ward and a "higher-level" ward consisting of all ward masters. The design assumes that inter-ward traffic will not be high enough to cause ward master intercommunications to become a bottleneck.

The *ward set* is the set of replicated data stored within the ward. Selective replication [9, 7] allows each ward member to store selected portions of the volume. Thus, the ward set may be smaller than the complete volume. The data stored by a particular volume replica is its *replica set*; the ward set is the union of all replica sets for all ward members.

The ward set changes dynamically, expanding and contracting as individual ward members add or drop particular files. Also, as machines storing replicas move between wards, ward sets change.

Roam does not currently contain a scheme for electing new ward masters, but the problem is similar to many other distributed system election problems, and similar solutions could be used. Roam can use a cheap solution that does not necessarily guarantee that only one leader will be present at a time, since the system can operate correctly with any number of wards and ward masters. Excess wards can be coalesced when convenient.

3.2 Synchronization

Consistency is maintained simultaneously within each ward and among wards. *Consistency topologies* specify which replicas exchange updates. The correctness of our consistency algorithms is topology-independent, assuming the topology allows information flow from any replica to any other replica, regardless of ward membership.

The current implementation uses an *adaptive ring* topology, both within and among wards. Briefly, the topology specifies a circular pattern of preferred reconciliation partners. If the preferred partner is unavailable, the topology specifies alternative partners. A ring is a good topology for small numbers of wards, since it limits overall costs while still providing reasonably fast dissemination of updates. For higher scales, a different reconciliation topology would be preferable [15]. A good reconciliation topology will lighten a ward master's load by limiting responsibility for propagating updates to a small number of other ward masters.

3.3 Support for Mobility

Mobile machines can move in several ways. Portable machines can be moved around in a small area, such as within a single suite of offices linked by a wireless LAN. Travelers can take their machines on short trips to distant locations. Longer trips, including permanent relocations, can cause long-term changes in the location of replicas. Roam has different solutions for each class of mobility.

Intra-ward mobility occurs when the user is mobile within the restricted geographic area covering his current ward. Moving within one's office from the desk to the couch is an intra-ward mobile action. Intra-ward mobility is important because it happens frequently.

The Ward Model assumes that intra-ward mobility is the dominant form of mobility, and makes it free. Since any replica in a ward can communicate directly with any other replica, the system does not even notice this form of mobility.

Temporary *inter-ward mobility* should be lightweight and inexpensive. Users will generally accept sub-optimal performance, given that the motion is temporary and the up-front cost is minimal. On the other hand, users moving for longer periods of time are generally willing to pay a larger up-front cost to gain better long-term performance.

Roam includes separate support for these two forms of inter-ward mobility. *Ward changing* is used during long stays, while *overlapping* is used during visits.

Ward changing involves physically changing the mobile replica's “home” ward, resulting in changes to the ward membership information in both the old and new wards. The ward set in the new ward might expand if the new member stores files not previously in the ward set. Similarly, the ward set at the old ward may shrink.

Since both old and new ward sets potentially change, and these changes are eventually propagated to other ward masters, ward changing can be a heavyweight operation. However, users benefit from the up-front cost in that all local data can be synchronized completely within the local ward.

Ward overlapping is a very lightweight mechanism with minimal up-front cost. The ward set of the new ward not changed, causing no global changes within the system. Only the new ward is affected by the operation; the information concerning the addition of a new member must be propagated to the other members. The old ward is not notified of the change, and the mobile replica is merely treated as any other temporarily unavailable replica.

Ward overlapping is implemented using simultaneous multi-ward membership, enabling direct communication with the members of each ward. We avoid changing the ward set of the new ward by making the new replica an “overlapped” member. For overlapped members, we do not merge the existing ward set with the mobile machine's replica set. Files in the intersection of the replica set and the ward set can be reconciled locally in the new ward. Files in the set difference must either temporarily remain unsynchronized or be reconciled with the original home ward.

4. Maintenance of Consistency

We call our replica synchronization process *reconciliation*. Reconciliation never directly involves more than two replicas.

Reconciliation is a pull-only process—new information is propagated in one direction. The pull-only strategy is simpler to model and describe. It is a more general approach, allowing support for one-way communication media, such as floppy disks. Often, models with significant round-trip latencies are better implemented as two one-way communication paths.

Reconciliation is separated into two major processes. The *recon* process uses meta-information from a remote replica to decide what local objects require updates. The *server* process handles data requests from the recon process, fetches file system data from remote replicas, and installs the actual file system modifications. The recon process is initiated on demand when either users or automated daemons initiate reconciliation. The server process is always resident on each participant host, though it is quiescent and cheap when inactive.

This division of labor is desirable for a number of reasons. First, it reduces complexity by separating functionality into two major categories: comparing replicas and sending and receiving data.

Second, the server is a common point that controls the transport of all information. This common point has potential security advantages and allows a *transport-independent* design, letting the recon process execute independently of the physical transport mechanism (email, `rshell`, TCP sockets, floppy-disk transfer) being used between machines.

The recon process executes a file-system scanning module at both the local and remote sites (via the server). The file-system scan updates the Roam data structures to reflect the current state of replicated objects. The recon process then compares the two states, sending data requests and file-system modifications to the server process, which asynchronously performs the operation. When the server indicates that all requests have been fulfilled, the recon process terminates.

More information on both processes can be found in [9].

5. Version Vector Maintenance

Version vectors [6] are used by Roam to track updates and compare data versions. Version vectors are well known and proven correct. However, they do not scale well because each replica maintains its own dedicated position in the vector.

One method to address scalability would be to form the version vector on a per-ward basis, with ward masters managing the inter-ward details. However, doing so means that moving from one ward to another would involve participation by the old and new ward masters, violating our initial invariant banning pre-motion actions.

We therefore studied the version vector, and noted two key observations:

- Updates typically occur in a few isolated “hot-spots.”
- Once all replicas have the same element for replica R, replica R's element is no longer relevant for version vector comparison purposes.

Thus, dynamic version vector maintenance would dramatically increase its scalability:

- Rather than preallocating a vector element for each replica R, we can dynamically expand the vector and create R's element when R generates its first update.
- Once all replicas have the same value for replica R_j in their vector, R_j's position can be removed or compressed from each vector with no loss of distinguishing information. Should replica R_j generate future updates, it can re-expand the vector via dynamic expansion.

Dynamic version vector compression on R_j’s element is challenging if anyone, including R_j, can simultaneously generate updates. Even temporarily blocking R_j from generating updates while executing a distributed algorithm to remove its element would nullify the whole advantage of optimistic replication.

We never store zero elements in the version vector, and we dynamically expand the vector when a replica generates its first update. Replicas that never generate updates are never mentioned in the version vector. This approach is not sufficient to solve the scalability problem, since many replicas may eventually generate updates, but it is useful in conjunction with dynamic compression.

Studies of replicated file systems [10] illustrate that conflict rates are generally quite low, meaning that concurrent writing of the same object (within the synchronization time-window) rarely occurs. Objects tend to have "hot-spots" within the set of replicas. While the hot-spots may change over time, widely replicated objects are rarely consistently updated by everyone.

While we must provide the ability for everyone to generate updates, rarely does a significant percentage of the replicas update at the same time. This observation suggests that the elements for the *cold* replicas, those not in the current hot-spot, play only a minor role in the version vector. On the (by definition) rare occasions when a cold replica generates an update, its element quickly stabilizes at all replicas, since the element does not change again soon. Once the element stabilizes, it no longer plays an important or distinguishing role in the version vector comparison algorithm and can be removed.

We periodically execute a distributed algorithm to remove the elements generated by the cold replicas. During the element-removal process we still must retain the high availability of optimistic replication systems. Therefore, we allow any replica, including the one whose element is being removed, to continue generating updates. We do so by forming consensus on a value to be subtracted from the element. At the end of consensus, if the value to be subtracted is equivalent to the element itself, then the result after subtraction is zero and the entire element is removed. Otherwise, the value of the element is simply reduced in ordinal value.

Full details on this algorithm can be found in [9].

6. Performance

We studied the disk space overhead and synchronization time to demonstrate Roam's cost and Roam's scalability. We also measured the cost of ward motion. Additional experiments and measurements can be found in [9, 7].

6.1 Disk Space Overhead

Roam stores its non-volatile data structures in look-aside databases on disk. Minimal disk overhead is an important and visible performance criterion. Additionally, Roam should support hundreds of replicas with minimal impact on disk overhead.

We measured the disk overhead of Roam by creating multiple wards. We compared Roam's disk overhead to disk overhead in the Rumor file system, as Rumor is the only other user-level peer-replication package available. (See Section 7 and [11] for more details on Rumor.) We computed disk overhead using the Linux `du` program, which measures file size in 1KB disk blocks. The test volume consists of 307 files, of which 38 are directories, totaling 13.6MB of data. Some directories have only a few files; the volume root directory has 151. The largest file is 1.4MB, and some files are only a few bytes long.

Figure 1 shows disk overhead as a percentage of total user data, as the number of replicas increases. For the Roam measurements, new wards are created every four replicas: replicas 1 to 3 belong to ward one, replicas 4 to 7 belong to ward two, etc. The overhead is always measured at replica 2, which is not a ward master.

The Rumor curve shows that users pay an approximate 2% overhead when creating the first replica, and the overhead increases linearly, less than .1% per additional replica. However, Roam's overhead only increases linearly as we create new replicas within the measured ward. The creation of new replicas in other wards does not add disk overhead in the first ward. The addition of each new ward causes a small increase in each replica's disk overhead, since each replica stores the identities of the other ward masters.

The per-ward overhead is dramatically smaller than the per-replica overhead. Therefore, Roam will scale well if no single ward ever becomes too large.

6.2 Synchronization Performance

The synchronization experiments used two portable machines connected by a quiet 10MB Ethernet. In all cases, we minimized the daemons and processes on the two machines. The first machine is a Dell Latitude XP with a 486DX4 running at 100Mhz with 36MB of main memory. The second is a TI TravelMate 6030 with a 133Mhz Pentium and 64MB of main memory. Reconciliation was always performed transferring data from the Dell machine to the TI machine.

We varied the amount of data updated from 0 to 100%, and within each trial we randomly selected a set of updated files. Since the files were selected at random, a given figure of *X%* is only an approximation of the amount of data updated. We used the same 13.6MB volume as above and performed at least seven trials at each data point.

The results are illustrated in Figure 2 where 95% confidence intervals are indicated. Rumor problems prevented reconciliation from completing with 100% modifications on this volume, so a data point corresponding to 100% modified using Rumor is not shown.

Figure 2 shows some important details. First, it shows that Roam is slightly, but not dramatically, more costly than Rumor. The performance slowdown is entirely a CPU cost, not extra transmission time over the network. Roam's system time remains constant while Rumor's system time increases linearly with the amount of data transferred.

At the 0% level, Rumor and Roam spend nearly identical amounts of time in the kernel; however, at the 25%, 50%, and 75% levels, Rumor spends approximately 3, 3.5, and 5 times more time in the kernel than Roam, respectively.

The difference arises from the data transport method. Rumor's method spends more time in system calls and in the kernel moving buffers around than Roam's method. As a result, Rumor doesn't scale well when increasing amounts of data are transferred. Under a “real” workload, when the other daemons and processes are not present, Rumor's kernel requirements would have a significant affect on the overall performance.

Figure 2 also provides insights into Roam's scalability. Reconciliation performance typically degrades in proportion to the number of replicas, given increased data structure size [9]. However, we expected that wards and ward masters would largely isolate the replicas from the increased effect of scale.

To test our hypothesis, we created 64 replicas of the preceding 13.6MB volume. Rather than collecting 64 separate portables, interconnecting them and creating a replica on each, we achieved the same effect by creating multiple wards on two 200Mhz Pentium Pro server machines. Six wards of ten members each were created on these two machines. The seventh ward contains four replicas on real portable machines, allowing easy comparison with our other measurements. We therefore simulated the effect of 64 replicas across 7 wards. The machines are real; in many cases, the network is simulated. However, we can still measure the performance between two portable machines that are connected by a real physical network.

Synchronization performance between two replicas in the same ward remains statistically unchanged as we move from a 1-ward, 3-replica system to a 7-ward, 64-replica system. The ward master isolates ward members from the effects of large scale; the members' data structures do not reflect the additional wards, so their synchronization performance is not affected by more wards.

Figure 2 demonstrates an important aspect of Roam's scalability. While the number of wards may increase, the synchronization performance within each ward remains constant within statistical error.

6.3 Ward Motion

Ward overlapping is intended to be a lightweight, temporary form of motion that is easy to perform and undo. However, synchronization performance can become worse during overlapping. When the moving replica stores objects that are not part of the new ward, they must be synchronized with the original ward (or else remain unsynchronized during the overlap). Ward changing is a more heavyweight, permanent form of motion that provides optimal synchronization performance in the new ward.

Experiments illustrated that the user's cost of performing either initial operation (changing or overlapping) is almost equivalent [9]. Essentially, both require two one-way reconciliations (or one two-way exchange of information). The real difference in cost lies in the effects of the operation on the local replica and on the system as a whole.

Ward overlapping has two associated costs: the extra disk space to belong to multiple wards, and the extra reconciliation time required to synchronize with the old ward any files that are not stored within the new ward. The extra disk overhead is linearly proportional to the number of members in the new ward, as the dominant source of disk overhead comes from the on-disk directory structure used to store the information about wards and their participants.

The additional synchronization time depends on the actual replication patterns, how many files must be reconciled with the old ward, and the type of connection to the old ward. Experiments [9] indicated that whenever the overlapped replica has to query the old ward, it pays a small constant cost just to inquire about updates. The cost is largely independent of the transport mechanism, since the metadata transmitted during the querying phase is small. The replica also pays transmission time for any updates that it could not receive via synchronization with the new ward. When the new ward stores everything that is stored on the mobile replica, there is no synchronization penalty.

When ward changing is used, the ward set at the new ward potentially expands and the ward set at the old ward potentially shrinks. Both changes need to be propagated to the other ward masters. Since the changes propagate asynchronously and via gossiping, the associated costs are difficult to measure directly. Instead, we characterize the amount of changed data, rather than the time required to distribute the changes.

The ward set at the new ward expands by the number of new file objects brought in by the moving replica. For each file object added to the ward set, the ward master must store a 255-byte entry in its lookaside databases. If the ward master also stores an actual copy of the file data, it pays a further penalty equal to the number of bytes in the file.

The other ward masters must learn about the changes in the ward set; this requires propagating 255 bytes per new object to them. The changes are propagated to the ward masters in a need-to-know fashion, with no cost placed on ward masters that don't care about the given change; the worst case occurs when all ward masters must know.

If we assume the ward set increases by n objects, then 255 * n bytes must be distributed to all other ward masters. An increase in 100 files causes an overhead of only 25.5KB. Nevertheless, the overhead must be distributed to all ward masters. Given m ward masters, we effectively introduce a network overhead of 255 * n * (m-1) bytes. Assuming 50 ward masters and our previous 100-file increase, the worst-case network overhead amounts to 1.25MB.

The actual network cost depends on the reconciliation topology. Since we utilize gossip-based communication, each ward master could potentially add new information to the 255 * n-byte overhead. If both ward masters M_1 and M_2 were expanding their ward sets, and the expansions happened to be on the same n files, each ward master would simply add its new information to the existing 255 * n-byte overhead, rather than generating additional network overhead. Gossiping allows the costs to be combined in a non-additive manner.

Finally, since the overhead is distributed during normal reconciliation, each ward master potentially includes information about new updates. For example, if all n files had updates that needed to be distributed to the other ward masters, the extra overhead attributable to ward changing would be zero, since the information must be transfered anyway to maintain consistency.

7. Related Work

Coda [12] is an optimistically replicated file system primarily using the client-server model. Coda provides replication flexibility akin to selective replication at the clients, but not at the replicated servers. The servers run a form of peer replication. Coda clients cannot directly inter-communicate due to restrictions of the client-server model. Use of this mode dramatically simplifies the consistency algorithms, at the cost of limiting the system's utility for mobility.

Coda is clearly superior in the low-bandwidth scenario, having greatly optimized communications and synchronization, especially in environments with weak connectivity [4]. Some of the same ideas could be applied in Roam, though additional research would be required to incorporate them into a peer model.

The Little Work project [2] is similar to Coda, but modifies only the clients, leaving the AFS [3] servers unaltered. Congestion caused by clients' slow links is reduced in a variety of ways, including client-side modifications of AFS, Little Work's underlying RPC, and other congestion-avoidance and control methods. However, clients cannot directly communicate, hindering the usability of the system in dynamic mobile environments.

The Bayou system [14] replicates databases rather than file system objects. Like Roam, it uses the peer-to-peer model. Unlike Roam, Bayou does not attempt to provide transparent conflict detection. Applications must specify a condition that determines when a conflicting access has been made, and must specify the particular resolution process.

Ficus [5] is one of the intellectual predecessors of Roam, and Roam therefore shares many of its characteristics. Both are based on a peer model, though Roam's Ward Model scales better than the Ficus traditional peer model. Both provide selective replication control. While each maintains consistency with a periodic reconciliation process, Ficus also uses a best-effort propagation at update time.

Ficus is aimed at a distributed Internet environment, and works well for its target. However, it is unsuitable for mobile use, and does not scale well.

Rumor [11] is the direct predecessor of Roam; much of Roam's implementation is directly based on modified Rumor code. Rumor is, in turn, a descendent of Ficus, and shares many of its characteristics and problems. It is based on the traditional peer model, and relies upon periodic reconciliation to maintain consistency. While Rumor is better suited to a mobile environment than Ficus, its scaling properties are substantially the same.

8. Conclusion

Existing replication systems are not designed to handle some important mobile scenarios. True mobile computing has only recently become affordable and feasible; previous solutions were designed without considering the ramifications and possible effects of physical motion on the replication system. Roam builds upon previous work by defining a new replication model and designing a replication system to support real mobile users.

Roam provides a scalable, any-to-any communication model. It gains efficiency through selective replication, and provides users with a “mobile anywhere, anytime” framework. Roam has been implemented and tested.

The concepts and ideas behind Roam are applicable beyond the realm of file replication. The Ward Model may apply well in other large-scale environments that share common characteristics, such as in the mobile IP problem, network routing protocols, and Internet security. The algorithms behind version vector compression also apply to more general distributed consensus-forming problems. The selective-replication protocols and algorithms also apply to more general problems of managing individual objects within a large container.

References

[1] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 1173-182. ACM, June 1996.

[2] P. Honeyman, L. Huston, J. Rees, and D. Bachmann. The Little Work Project. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 11-14. IEEE, April 1992.

[3] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51-81, February 1988.

[4] J. J. Kistler and M. Satyanarayanan. Disconnected Operations in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3-25, 1992.

[5] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on Optimistically Replicated, Peer-to-Peer Filing. *Software—Practice and Experience*, 27(12), December 1997.

[6] D. S. Parker, Jr., G. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Wards, S. Kiser, and C. Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, 9(3): 240-247, May 1983.

[7] D. Ratner, G. Popek, and P. Reiher. Peer Replication With Selective Control. UCLA Computer Science Department Technical Report CSD-960031, July 1996.

[8] D. Ratner, P. Reiher, G. J. Popek, and G. H. Kuenning. Replication Requirements in Mobile Environments. Presented at the *First Dial M for Mobility* conference, October 1997.

[9] D. H. Ratner. Roam: A Scalable Replication System for Mobile and Distributed Computing. PhD thesis, University of California, Los Angeles, CA, January 1998. Also available as UCLA CSD Technical Report UCLA-CSD-970044.

[10] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving File Conflicts in the Ficus File System. In *USENIX Conference Proceedings*, pages 183-195. University of California, Los Angeles, USENIX, June 1994.

[11] P. Reiher, J. Popek, M. Gunter, J. Salomone, and D. Ratner. Peer-to-Peer Reconciliation Based Replication for Mobile Computers. In *Proceedings of the ECOOP Workshop on Mobility and Replication*, July 1996.

[12] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers,* 39(4):447-459, April 1990.

[13] M. Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, and M. West. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, December, 1985.

[14] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th Symposium on Operating Systems Principle*s, pages 172-183. Copper Mountain Resort, Colorado, December 1995.

[15] A. Wang, P. Reiher, R. Bagrodia, and G. Popek. A Simulation Evaluation of Optimistic Replicated Filing in a Mobile Environment. *18th IEEE International Performance, Computing, and Communications Conference*, February 1999.