Replication Requirements in Mobile Environments *

David Ratner [a], Peter Reiher [b], Gerald J. Popek [b,**], and Geoffrey H. Kuenning [c]

[a] *Software.com, Inc., Santa Barbara, CA 93103, USA*
E-mail: dratner@excite.com

[b] *Department of Computer Science, University of California, Los Angeles, CA 90095, USA*
E-mail: {reiher,popek}@cs.ucla.edu

[c] *Department of Computer Science, Harvey Mudd College, Claremont, CA 91711, USA*
E-mail: geoff@cs.hmc.edu

Replication is extremely important in mobile environments because nomadic users require local copies of important data. However, today's replication systems are not "mobile-ready." Instead of improving the mobile user's environment, the replication system actually hinders mobility and complicates mobile operation. Designed for stationary environments, the replication services do not and cannot provide mobile users with the capabilities they require. Replication in mobile environments requires fundamentally different solutions than those previously proposed, because nomadicity presents a fundamentally new and different computing paradigm. Here we outline the requirements that mobility places on the replication service, and briefly describe ROAM, a system designed to meet those requirements.

1. Introduction

Mobile computing is rapidly becoming standard in all types of environments: academic, commercial, and private. Widespread mobility impacts multiple arenas, but one of particular importance is data replication. Replication is especially important in mobile environments, since disconnected or poorly connected machines must rely primarily on local resources. The monetary costs of communication when mobile, combined with the lower bandwidth, higher latency, and reduced availability, effectively require that important data be stored locally on the mobile machine. In the case of shared data, between multiple mobile users or between mobile and stationary machines, replication is often the best and sometimes the only viable approach.

Many replication solutions [4,16] assume a static infrastructure; that is, the connections themselves may be transient but the connection *location* and the set of possible synchronization partners always remain the same. However, mobile users are by definition not static, and a replication service that forces them to adjust to a static infrastructure hinders mobility rather than enables it. Extraordinary actions, such as long distance telephone calls over low-bandwidth links, are necessary for users to conform to the underlying static model, costing additional time and money while providing a degraded service. Additionally, mobile users have difficulty interoperating with other mobile users, because communication patterns and topologies are typically predefined according to the underlying infrastructure. Often, direct synchronization between mobile users is simply not permitted.

Other systems [2,14,18] have simply traded the above communication problem for another one: scaling. They provide the ability for *any-to-any* synchronization, but their model suffers from inherent scaling problems, limiting its usability in real environments. Good scaling behavior is

* This work was sponsored by the Advanced Research Projects Agency under contract DABT63-94-C-0080.

** Gerald Popek is also affiliated with CarsDirect.com.

very important in the mobile scenario. Mobile users clearly require local replicas on their mobile machines. Yet, replicas must also be stored in the office environment for reliability, intra-office use by non-mobile personnel, and system-administration activities like backups. Additionally, typical methods for reducing replication factors, such as local area network sharing techniques, are simply not feasible in the mobile context. Mobile users require local replicas of critical information, and in most cases desire local access to non-critical objects as well, for cost and performance reasons. The inability to scale well is as large an obstacle to the mobile user as the restriction of a static infrastructure discussed above.

The main problem is that mobile users are replicating data using systems that were not designed for mobility. As such, instead of the replication system improving the state of mobile computing, it actually hinders mobility, as users find themselves forced to adjust their physical motion and computing needs to better match what the system expects. This paper outlines the requirements of a replication service designed for the mobile context. We conclude with a description of ROAM, a replication solution redesigned especially for mobile computing. Built using the Ward architecture [11], it enables rather than hinders mobility, and provides a replication environment truly suited to mobile environments.

2. Replication Requirements

Mobile users have special requirements above and beyond those of simple replication required by anyone wishing to share data. Here we discuss some of the requirements that are particular to mobile use: any-to-any communication, larger replication factors, detailed controls over replication behavior, and the lack of pre-motion actions. We omit discussion of well-understood ideas, such as the case for optimistic replication, discussed in [2,3,5,17].

2.1. Any-to-any communication

By definition, mobile users change their geographic location. As such, it cannot be predicted *a priori* what machines will be geographically collocated at any given time. Given that it is typically cheaper, faster, and more efficient to communicate with a local partner rather than a remote one, mobile users want the ability to directly communicate and synchronize with whomever is "nearby." Consistency can be correctly maintained even if two machines cannot directly synchronize with each other, as demonstrated by systems based on the client-server model [4,16], but local synchronization increases usability and the level of functionality while decreasing the inherent synchronization cost. Users who are geographically collocated don't want updates to *eventually* propagate through a long-distance, sub-optimal path: the two machines are next to each other, and the synchronization should be instantaneous.

Since users expect that nearby machines should synchronize with each other quickly and efficiently, and it cannot be predicted which machines will be geographically collocated at any point in the future, a replication model capable of supporting *any-to-any communication* is required. That is, the model must allow any machine to communicate with any other machine—there can be no second-class clients in the system.

Any-to-any communication is also required in other mobile arenas, such as in *appliance* mobility [6], the motion from device to device or system to system. For instance, given a desktop, a laptop, and a palmtop, it is unlikely that one would want to impose a strict client-server relationship between the three; rather, one would want each to be able to communicate with any of the others.

Providing any-to-any communication is equivalent to using a *peer-to-peer* replication model [10, 14,18]; if anyone can directly synchronize with anyone else, then everyone must by definition be equals, at least with respect to update-generation abilities. Some, however, have argued against peer models in mobile environments because of the relative insecurity regarding the physical devices themselves—for example, laptops are often stolen. The argument is that since mobile computers are physically less secure, they should be "second-class" citizens with respect to the highly secure servers located behind locked doors [15]. The class-based distinction is intended to provide improved security by limiting the potential security breach to only a second-class object.

The argument is based on the assumption that security features must be encapsulated within the peer model, and therefore unauthorized access to any peer thwarts all security barriers and mechanisms. However, systems such as TRUFFLES [13] have demonstrated that security policies can be modularized and logically situated around a peer replication framework while still remaining independent of the replication system. TRUFFLES, an extension to the peer-based systems FICUS [2] and RUMOR [14], incorporates encryption-based authentication and over-the-wire privacy and integrity services to increase a replica's confidence in its peers. TRUFFLES further supports protected definition and modification of security policies. For example, part of the security policy could be to only accept new file versions from specific (authenticated) replicas—which is effectively the degree of security provided by the "second-class" replicas mentioned above.

With such an architecture, the problems caused by unauthorized access to a peer replica are no different from the unauthorized access of a client in a client-server model. Thus, the question of update-exchange topologies (any-to-any as compared to a more stylized, rigid structure as in client-server models) can be dealt with independently of the security issue and the question of how to enforce proper security controls.

2.2. Larger replication factors

Most replication systems only provide for a handful of replicas of any given object. Additionally, peer algorithms have never traditionally scaled well. Finally, some have argued that peer solutions simply by their nature cannot scale well [15].

However, while mobile environments seem to require a peer-based solution (described above), they also seem to negate the assumption that a handful of replicas is enough. While we do not claim a need for thousands of writable copies, it does seem likely that the environments common today and envisioned for the near future will require larger replication factors than current systems allow.

First and foremost, each mobile user requires a local replica on their laptop, doubling replication factors when data is stored both on the user's desktop and laptop. Additionally, although replication factors can often be minimized in office environments due to LAN-style sharing and remote-access capabilities, such network-based file sharing cannot be utilized in mobile environments due to the frequency of network partitions and the wide range of available bandwidth and transfer latency.

Second, consider the case of appliance mobility. The above discussion assumes that each user has one static machine and one mobile machine. The future will see the use of many more "smart" devices capable of storing replicated data. Palmtop computers are becoming more common, and there is even a wristwatch that can download calendar data from another machine. Researchers [19] have built systems that allow laptop and palmtop machines to share data dynamically and opportunistically. It is not difficult to imagine other devices in the near future having the capability to store and conceivably update replicated data; such devices potentially increase replication factors dramatically.

Finally, some have argued the need for larger replication factors independent of the mobile scenario, such as in the case of air traffic control [9]. Other scenarios possibly requiring larger replication factors include stock exchanges, network routing, airline reservation systems, and military command and control.

Read-only strategies and other class-based techniques cannot adequately solve the scaling problem, at least in the mobile scenario. Class-based solutions are not applicable to mobility, for the reasons described above (Section 2.1). Read-only strategies are not viable solutions because they force users to pre-select the writable replicas beforehand and limit the number of writable copies. In general one cannot predict which replicas require write-access and which ones do not. We must provide the *ability* for all replicas to generate updates, even though some may never do so.

2.3. Detailed replication controls

By definition, a replication service provides users with some degree of replication *control*—a method of indicating what objects they want replicated. Many systems provide replication on a large-granularity basis, meaning that users requiring one portion of the container must locally replicate the entire container. Such systems are perhaps adequate in stationary environments, when users have access to large disk pools and network resources, but replication control becomes vastly more important to mobile users. Nomadic users do not in general have access to off-machine resources, and therefore objects that are not locally stored are effectively inaccessible. Everything the user requires must be replicated locally, which becomes problematic when the container is large.

Replicating a large-granularity container means that some of the replicated objects will be deemed unimportant to the particular user. Unimportant data occupies otherwise usable disk space, which cannot be used for more critical objects. In the mobile context, where network disconnections are commonplace, important data that cannot be stored locally causes problems ranging from minor inconveniences to complete stoppages of work and productivity, as described by Kuenning [7]. Kuenning's studies of user behavior indicate that the set of required data can in fact be completely stored locally, but only if the underlying replication service provides the appropriate flexibility to individually select objects for replication. Users and automated tools therefore require fairly detailed controls over what objects are replicated, because without them mobile users cannot adequately function.

2.4. Pre-motion actions

One possible design point would have users "register" themselves as nomads for a specific time duration before becoming mobile. In doing so, the control structures and algorithms of the replication system could be greatly simplified; users would act as if they were stationary, and register their motion as the unusual case. For instance, suppose a user was taking a three-day trip from Los Angeles to New York. Before traveling, machines in Los Angeles and New York could exchange state to "re-configure" the user's portable to correctly interact with the machines in New York. Since replication requires underlying distributed algorithms, part of the reconfiguration process would require changing and saving the distributed state stored on the portable, to ensure correct algorithm execution.

However, such a design policy drastically restricts the way in which mobility can occur, and does not match with the reality of mobile use. Mobility cannot always be predicted or scheduled. Often the chaos of real life causes unpredicted mobility: the car fails *en route* to work, freeway traffic causes unforeseeable delays, a child has to be picked up early from school, a family emergency occurs, or weather delays travel plans. Users are often forced to become mobile earlier or remain mobile longer than they had initially intended. In general, we cannot require that users know *a priori* either when they will become mobile or for how long.

Additionally, this design policy makes underlying assumptions about the connectivity and accessibility of machines in the two affected geographic areas: Los Angeles and New York, in the above example. It assumes that before mobility occurs, the necessary machines are all accessible so the state-transformation operation can occur. Inaccessibility of any participant in this process blocks the user's mobility. Such a policy seems overly restrictive, and does not match the reality of mobile use. Perhaps a user wants to change geographic locations precisely *because* a local machine is unavailable, or perhaps a user needs to become mobile at an instant when connectivity is down between the multiple required sites. Since neither mobility nor connectivity can be predicted, one cannot make assumptions on the combination of the two.

For these reasons, we believe that solutions that require "pre-motion" actions are not viable in the mobile scenario. Pre-motion actions force users to adapt to the system rather than having the system support the desired user behavior. Any real solution must provide the type of "get-up and go" functionality required by people for everyday use.

3. Roam

ROAM is a system designed to meet the above set of requirements. It is based on the *Ward* model [11] and is currently being implemented and tested at the University of California at Los Angeles.

3.1. Ward model

The Ward model combines classical elements of both the traditional peer-to-peer and client-server models, yielding a solution that scales well and provides replication flexibility, allowing dynamic reconfiguration of the synchronization topology. The model's main grouping mechanism is the *ward*, or Wide Area Replication Domain. A ward is a collection of "nearby" machines, possibly only loosely connected. The definition of "nearby" depends on factors such as geographic location, expected network connectivity, bandwidth, latency, and cost; see [12] for a full discussion of these issues.

Wards are created as replicas are added to the system: each new replica chooses whether to join an existing ward or form a new one. We believe that it is possible to automate the assignment of ward membership, but as the issues involved are complex, we have avoided attempting to do so in the current system. Instead, this decision is controlled by a human, such as a system administrator or knowledgeable user. If necessary, the decision can be altered later by using ward-changing utilities.

Although all members of the ward are equal peers, the ward has a designated *ward master*, similar to a server in a client-server model but with several important differences:

- Since all ward members are peers, any two ward members can directly synchronize with one another. Typical client-server solutions do not allow client-to-client synchronization. Whether by design or by accident, mobile users will often encounter other mobile users; in such cases, direct access to the other ward member may be easier, cheaper and more efficient than access to the ward master.
- Since all ward members are peers, any ward member can serve as the ward master. Automatic re-election and *ward-master reconfiguration* can occur should the ward master fail or become unavailable, and algorithms exist to resolve multiple-master scenarios. Correctness is not affected by a transient ward master failure, but the system maintains better consistency if the ward master is typically available and accessible. Since neither an inaccessible ward master nor multiple ward masters affects overall system correctness (see Section 3.2), the re-election problem is considerably easier than related distributed re-election problems.
- The ward master is not required to store actual data for all intra-ward objects, though it must be able to identify (i.e. name) the complete set. Most client-server strategies force the server to store a superset of each client's data.

The ward master is the ward's only link with other wards; that is, only the ward master is aware of other replicas outside the ward. This is one manner in which the ward model achieves wards.idraw

Figure 1. The basic ward architecture. Overlapped members are a mobility feature (Section 3.4).

good scaling—by limiting the amount of knowledge stored at individual replicas. Traditional peer models force every replica to learn about other replica's existence; in the ward model, replicas are only knowledgeable about the other replicas within their own ward. In fact, most replicas are completely unaware of the very existence of other wards.

All ward masters belong to a higher-level ward, forming a two-level hierarchical model.[1] Ward masters act on their ward's behalf by bringing new updates into the ward, exporting others out of the ward, and gossiping about all known updates. Consistency is maintained across all replicas by having ward masters communicate directly with each other and allowing information to propagate independently within each ward. Figure 1 illustrates the basic architecture, as well as advanced features discussed in later sections.

Wards are dynamically formed when replicas are created, and are dynamically maintained as suitable ward-member candidates change. Ward destruction occurs automatically when the last replica in a given ward is destroyed.

3.2. System correctness

An important feature of the ward model is that system correctness does not depend on having precisely one master per ward. Even during reconfiguration, updated files will flow between replicas without loss of information or other incorrect behavior. For most purposes, the ward master is simply another replica. Whether communicating within its own ward or with other wards, the master maintains consistency using the same algorithms as the non-master replicas. Thus, the propagation of information within and among wards follows from the correctness of these algorithms, first described in [1].

If a ward master becomes temporarily unavailable, information will continue to propagate between other ward members, due to the peer model. However, information will not usually propagate to other wards until the master returns. An exception to this rule will occur if a ward member temporarily or permanently moves to another ward, as described in Section 3.4, carrying data with it.

If the master suffers a permanent failure, a new master can be elected. We must then demonstrate that correctness will not suffer during the transition to the new master. Correctness will be violated either if the failed master had some information that cannot be reconstructed, or if the failed master's participation is required for the completion of some distributed algorithm. The first case can occur only if the lost information had been created at the master and had not yet propagated to another replica. In this case, the lost information cannot affect correctness because the situation is the same as if it had never existed. The second case is handled by distributed failure-recovery algorithms that are invoked when an administrator declares the old master as unrecoverable.

If a new ward master is elected, there is a possibility of creating multiple masters. Correctness is not affected in this case because the master does not play any special role in the algorithms. The purpose of a ward master is not to coordinate the behavior of other ward members, but rather to serve as a conduit for information flow between wards. Multiple masters, like overlapped members (Section 3.4.2), will merely provide another communication path between wards. Since the peer-to-peer algorithms assume arbitrary communication patterns, correctness will not be affected by multiple ward masters.

3.3. Flexibility in the model

Replication flexibility is an important feature of the ward model. The set of data stored within each ward, called the *ward set*, is dynamically adjustable, as is the set of ward members themselves.

[1] The rationale behind the two-level hierarchy and its impact on scaling is discussed in Section 3.5.

As ward members change their data demands and alter what replicated data they store locally, the ward set changes. Similarly, as mobile machines join or leave the ward, the set of ward participants changes. Both the ward set and ward membership are locally recorded and are replicated in an optimistic fashion.

Additionally, each ward member, including the ward master, can locally store a different subset of the ward set. Such replication flexibility, called *selective replication* [10] provides improved efficiency and resource utilization: ward members locally store only those objects that they actively require. Replication decisions can be made manually or with automated tools [5,8].

Since the ward set varies dynamically, different wards might store different sets: not all ward sets will be equivalent. In essence, the model provides selective replication between wards themselves. The reconciliation topologies and algorithms [10] apply equally well within a single ward and between ward masters. Briefly, the algorithms provide that machines communicate with multiple partners to ensure that each data object is synchronized directly with another replica. Additionally, the data synchronization algorithms support the *reconciliation* of non-local data via a third-party data-storage site, allowing the ward master to reconcile data that is not stored locally but is stored somewhere within the ward.

3.4. Support for mobility

The model supports two types of mobility. *Intra-ward* mobility occurs when machines within the same ward become mobile within a limited geographic area; the machines encountered are all ward members. Since ward members are peers, direct communication is possible with any encountered machine. Intra-ward mobility might occur within a building, when traveling to a co-worker's house, or at a local coffee shop.

Perhaps more interesting, *inter-ward* mobility occurs when users travel (with their data) to another geographic region, encountering machines from another ward. Examples include businessmen traveling to remote offices and distant collaborators meeting at a common conference.

Inter-ward mobility raises two main issues. First, recall that due to the model's replication flexibility, two wards might not have identical ward sets. Thus, the mobile machine may store data objects not kept in the new ward, and vice-versa. Second, consider the typical patterns of mobility. Often users travel away from their "home location" for only a short time. The system would perform poorly if such transient mobile actions required global changes in data structures across multiple wards. On the other hand, mobile users occasionally spend long periods of time at other locations, either permanently or semi-permanently changing their definition of "home." In these scenarios, users should be provided with the same quality of service (in terms of local performance and time to synchronize data) as they experienced in their previous "home".

Our solution resolves both issues by defining two types of inter-ward mobility—short-term (transient) and long-term (semi-permanent)—and providing the ability to transparently and automatically upgrade from the former to the latter. The two operations are called *ward overlapping* and *ward changing* respectively. Collectively, the two are called *ward motion* and enable peer-to-peer communication between any two replicas in the ward model, regardless of their ward membership.

3.4.1. Ward changing

Ward changing involves a long-term, perhaps permanent, change in ward membership. The moving replica physically changes its notion of its "home" ward, forgetting all information from the previous ward; similarly, the other participants in the old and new wards alter their notion of current membership. Ward membership information is maintained using the same optimistic algorithms that are used for replicating data, so that the problem of tracking membership in often-disconnected environments is straightforward.

The addition of a new ward member may change the ward set. Since the ward master is responsible for the inter-ward synchronization of all data in the ward set, the ward set must expand to properly encompass the replicated data stored at the moving replica. Similarly, the ward set at the old ward may shrink in size, as the ward set is dynamically and optimistically recalculated when ward membership changes. The ward-set changes propagate to other ward masters in an optimistic, "need-to-know" fashion so that only the ward masters that care about the changes learn of them. Since both ward sets can potentially change, and these changes are eventually propagated to other ward masters, ward changing can be a heavyweight operation. However, users benefit because all local data can be synchronized completely within the local ward, giving users the best possible quality of service and reconciliation performance.

3.4.2. Ward overlapping

In contrast, ward overlapping is intended as a very lightweight mechanism, and causes no global changes within the system. Only the new ward is affected by the operation. The localization of changes makes it a lightweight operation both to perform and to undo.

Ward overlapping allows simultaneous multi-ward membership, enabling direct communication with the members of each ward. To make the mechanism lightweight, we avoid changing the ward sets by making the new replica an "overlapped" member instead of a full-fledged participant. Ward members (except for the ward master) cannot distinguish between real and overlapped members; the only difference is in the management of the ward set. Instead of merging the existing ward set with the data stored on the mobile machine, the ward set remains unaltered. Data shared between the mobile machine and ward set can be reconciled locally with members of the new ward. However, data outside the new ward cannot be reconciled locally, and must either temporarily remain unsynchronized or else be reconciled with the original home ward.

3.4.3. Ward motion summary

When a replica enters another ward, there are only two possibilities: the ward set can change or remain the same. The former creates a performance-improving but heavyweight solution; the latter causes a moderate performance degradation when synchronizing data not stored in the new ward but provides a very lightweight solution for transient mobile situations. Since both are operationally equivalent, the system can transparently upgrade from overlapping to changing if the motion seems more permanent than first expected.

Additionally, since ward formation is itself dynamic, users can easily form *mobile workgroups* by identifying a set of mobile replicas as a new (possibly temporary) ward. By using ward overlapping, mobile workgroups can be formed without leaving the old wards. Ward motion and dynamic ward formation and destruction allow easy and straightforward communication between any set of replicas in the entire system.

3.5. Scalability

The scalability of the ward model is directly related to the degree of replication flexibility. Ward sets can dynamically change in unpredictable ways; therefore the only method for a ward master to identify its ward set is to list each entry individually. The fully hierarchical generalization of the ward model to more than two levels faces scaling problems due to the physical problems of maintaining and indexing these lists of entries.

Nevertheless, the proposed model scales well within its intended environment, and allows several hundred read-write replicas of any given object, meeting the demands of everyone from a single developer or a medium-sized committee to a large, international company. The model could be adapted to scale better by restricting the degree of replication freedom. For instance, if ward sets changed only in very regular fashions, they could be named as a unit instead of naming all members, dramatically improving scalability. However, we believe that replication flexibility is an important design consideration in the targeted mobile environment, and one that users absolutely require, so we have chosen not to impose such regularity.

4. Performance

4.1. Disk space overhead

Roam, like Rumor before it, stores its non-volatile data structures in lookaside databases within the volume but hidden from the user. From the user's perspective, anything other than his or her actual data is overhead and effectively shrinks the size of the disk. Minimal disk overhead is therefore an important and visible criterion for user satisfaction.

Additionally, Roam is designed to be a scalable system. The Ward Model should support hundreds of replicas with minimal impact between wards. Specifically, the creation of a new replica in ward `X` should not affect the disk overhead of the replicas in other wards.

We therefore measured the disk overhead of Roam using two different volumes. The first of these volumes was chosen as a typical representative of a user's personal subtree, while the second was chosen to stress Roam by storing small files that would exaggerate the system's space overhead.

After empirically measuring the overhead under different conditions, we fitted equations to describe the overhead in terms of the number of files, types of files, number of replicas, and number of wards. These equations can be summarized as follows (full results are given in [12]):

- Each new directory costs 4.2KB + 30 bytes per object in the directory
- Each new file costs .24KB
- The first replica within the ward, even without any user data, costs 57.36KB
- Each additional replica within the ward costs 6.44KB + 12 bytes per object stored at the replica
- Each new ward costs 6.44KB

4.2. Synchronization performance

Since Roam's main task is the synchronization of data, we also measured the synchronization performance. We performed our experiments with two portable machines, in all cases minimizing extraneous processes to avoid non-repeatable effects. One machine was a Dell Latitude XP with a 486DX4 running at 100MHz with 36MB of main memory, while the second was a TI TravelMate 6030 with a 133MHz Pentium and 64MB of main memory. Reconciliation was always performed by transferring data from the Dell machine to the TI machine. In other words, the reconciliation process always executed on the TI machine.

Of course, reconciliation performance depends heavily on the sizes of the files that have been updated. Since Roam performs whole-file transfers, and any updated file must be transfered across the network in its entirety, we would expect reconciliation to take more time when more data has been updated. We therefore varied the amount of data updated from 0 to 100%, and within each trial we randomly selected the set of updated files. Since the files are selected at random, a given figure of `X`% is only an approximation of the amount of data updated, rather than an exact figure. In all measurements, we used the personal-subtree volume mentioned in Section 4.1, and performed at least seven trials at each data point.

We performed five different experiments under the above conditions. The first two compared Roam and Rumor synchronization performance over a 10MB quiet Ethernet and WaveLAN wireless cards respectively. The third studied the effect of increasing numbers of replicas; the fourth studied the effect of increasing numbers of wards. The fifth looked at the effects of selective replication [10] and different replication patterns on synchronization performance.

These experiments showed that Roam is 10%

to 25% slower than RUMOR when running with similar numbers of replicas. Most of the slowdown is due to ROAM's more flexible structure, which uses more processes and IPC to simplify the code and enhance scalability. Reconciliation of the 13.6MB volume under ROAM takes from 46 to 206 seconds, depending on the transport mechanism, the number of files modified, and the number of replicas in the ward.

We also studied the impact of multiple wards on the synchronization performance. We varied the number of wards from one, as in the previous experiments, to six. We placed three replicas within one of these wards, and measured the synchronization between two of them on the previously described portable machines. These experiments showed that at a 95% level of confidence, adding wards has no impact on synchronization performance between two replicas.

4.3. Scalability

We have already discussed some aspects of ROAM's scalability, such as in disk space overhead (Section 4.1). However, another major aspect of scalability is the ability to create many replicas and still have the system perform well during synchronization. Synchronization performance includes two related issues. First, the reconciliation time for a given replica in a given ward should be largely unaffected by the total number of replicas and wards. Second, the time to distribute an update from any replica to any other replica should presumably be faster in the Ward Model than in a standard approach (like RUMOR), or else we have failed in our task.

4.3.1. Reconciliation time

To measure the behavior of reconciliation time as the total number of replicas increases, we used a hybrid simulation. We created 64 replicas of our test volume, reducing the hardware requirements by using servers to store wards and replicas that were not actively participating in the experiments. Again, we found that at a 95% level of confidence, the synchronization time does not change as the system configuration was varied from one ward with a total of three replicas to 7 wards with a total of 64 replicas.

4.3.2. Update distribution

Another aspect of scalability concerns the distribution of updates to all replicas. A scalable system would presumably deliver updates to all replicas more quickly than a non-scalable system, at least at large numbers of replicas. Additionally, while it may not perform better at small numbers of replicas, a scalable system should at least not perform worse.

Rather than measuring elapsed time, which depends on many complicated factors such as connectivity, network partitions, available machines, and reconciliation intervals, we considered the number of individual, pair-wise reconciliation actions, and analytically developed equations that characterize the distribution of updates. We assume that there are $\mathtt{M}$ replicas; one of them, replica $\mathtt{R}$, generates an update that must propagate to all other replicas. The following equations identify the number of separate reconciliation actions that must occur, both on the average and in the worst case, to propagate the update from $\mathtt{R}$ to some other replica $\mathtt{S}$.

In a non-ward system such as RUMOR, since there are $\mathtt{M}$ replicas, $\mathtt{M}-1$ of which do not yet have the update, and reconciliation uses a ring between all $\mathtt{M}$ replicas, we need $\frac{\mathtt{M}-1}{2}$ reconciliation actions on average. The worst case requires $\mathtt{M}-1$ reconciliation actions.

The analysis for ROAM is a little more complicated. Assume that the $\mathtt{M}$ replicas are divided into $\mathtt{N}$ wards such that each ward has $\mathtt{M}/\mathtt{N}$ members. Propagating an update from $\mathtt{R}$ to $\mathtt{S}$ requires first sending it from $\mathtt{R}$ to $\mathtt{R}$'s ward master, then sending it from $\mathtt{R}$'s ward master to $\mathtt{S}$'s ward master, and then finally to $\mathtt{S}$. Of course, if $\mathtt{R}$ and $\mathtt{S}$ are members of the same ward, then much of the expense is saved; however, we will solve the general problem first before discussing the special case.

Under the above conditions, we need $\frac{1}{2}(\frac{\mathtt{M}}{\mathtt{N}}-1)$

reconciliation actions on average to distribute the update between a replica and its ward master, and $\frac{1}{2}(\mathtt{N}-1)$ actions on average between ward masters. From these building blocks, we calculate that, on average, ROAM requires the following number of reconciliation actions:

$$\begin{aligned}
&\frac{1}{2}(\frac{\mathtt{M}}{\mathtt{N}}-1)+\frac{1}{2}(\mathtt{N}-1)+\frac{1}{2}(\frac{\mathtt{M}}{\mathtt{N}}-1) \\
&=\frac{\mathtt{M}}{\mathtt{N}}-1+\frac{1}{2}(\mathtt{N}-1) \\
&=\frac{\mathtt{M}}{\mathtt{N}}+\frac{(\mathtt{N}-3)}{2} \qquad (1)
\end{aligned}$$

Note that when $\mathtt{N}=\mathtt{M}$, Equation 1 becomes $\frac{\mathtt{M}-1}{2}$ (RUMOR's performance). Setting $\mathtt{N}=\mathtt{M}$ eliminates any benefit from grouping. However, it is also interesting to note that when $\mathtt{N}=2$, Equation 1 *also* becomes $\frac{\mathtt{M}-1}{2}$. Having only two wards does not improve the required time to distribute updates (although it does improve other aspects such as data structure size and network utilization).

In general, ROAM distributes updates faster than RUMOR when $2<\mathtt{N}<\mathtt{M}$ and $\mathtt{M}>3$; otherwise, ROAM performs the same as RUMOR (with respect to update distribution). From the two equations we calculate that the optimal number of wards for a given value of $\mathtt{M}$ is $\mathtt{N}=\sqrt{2\mathtt{M}}$. The above conditions yield a factor of three improvement at 50 replicas, and a factor of five at 200 replicas. With a multi-level implementation, larger degrees of improvement are possible.

The analysis for ROAM also indicates that, in the worst case, ROAM requires $\frac{2\mathtt{M}}{\mathtt{N}}+\mathtt{N}-3$ reconciliation actions.

As a special case, if $\mathtt{R}$ and $\mathtt{S}$ are in the same ward, only $\frac{1}{2}(\frac{\mathtt{M}}{\mathtt{N}}-1)$ reconciliation actions are required on average, and $\frac{\mathtt{M}}{\mathtt{N}}-1$ in the worst case.

4.4. Ward motion

Recall from Section 3.4 that ROAM supports two different flavors of ward motion: overlapping and changing. Overlapping is 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 presumably short time period). Changing is a more heavyweight, permanent form of motion that costs more but provides optimal synchronization performance in the new ward.

We experimentally investigated the costs of both forms of ward motion, using the same 13.6MB volume used in the other tests. There are four types of costs involved in these operations:

1. Initial setup costs at the moving replica,
2. Disk overhead at the moving replica,
3. Costs imposed on other wards and ward members, and
4. Ongoing costs of synchronization.

We summarize our results here; complete data is given in [12].

We found that setting up either type of motion took from 60 to 80 seconds, depending on the number of files stored on the local machine. Somewhat surprisingly, ward changing required only about 7% more elapsed time than overlapping.

The disk overhead at the moving replica depends on the size of the destination ward. In essence, the other members of the destination ward must be tracked as if they were members of the replica's original ward, occupying 6.44KB plus 12 bytes per file (see Section 4.1). For ward overlapping, this cost must be paid for both the original ward's members and the destination ward's members, while for ward changing, only the destination's members must be tracked. In either case, these costs are insignificant compared to the space required by the volume itself.

The costs imposed on other replicas and wards are minimal for ward overlapping, but for ward changing the old and new ward masters must change their ward sets, and these differences will need to be propagated to all other ward masters by gossiping. However, the amount of information that must propagate is minimal (about 255 bytes per file that changes ward membership), so the additional network load is still quite low.

Finally, when wards are overlapped, synchronization costs increase because the moving replica must communicate with both the new ward and the old one. Synchronization with the (temporary) new ward will take about the same amount of time as it would have in the original ward. However, to synchronize any files not available in the new ward, the original must be contacted. We measured these additional costs for various replication patterns in our test volume (described in [12]). To simulate the fact that communication with the original ward is probably long-distance and thus slower, we used a WaveLAN network for these experiments. We found that the synchronization time depends on both the number of files found only in the original ward, and on the number of modified files. When no files had been modified, the time was essentially constant at about 45 seconds. By contrast, when 100% of the (locally-stored) files had been modified, synchronization took from 78 to 169 seconds, depending on the exact set of files shared between the two ward masters.

5. Conclusion

Replication is required for mobile computing, but today's replication services do not provide the key features required by mobile users. Nomadicity requires a replication solution that provides any-to-any communication in a scalable fashion with sufficiently detailed control over the replication decisions. Roam was designed and implemented to meet these goals, paving the way not just to improved mobile computing but to new and better avenues of mobile research. Performance experiments have shown that Roam is indeed scalable and can handle the mobility patterns expected to be displayed by future users.

References

[1] R. G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System.* Ph.D. dissertation, University of California, Los Angeles, June 1991. Also available as UCLA technical report CSD-910018.

[2] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71, Anaheim, CA, June 1990. USENIX.

[3] J. S. Heidemann, T. W. Page, Jr., R. G. Guy, and G. J. Popek. Primarily disconnected operation: Experiences with Ficus. In *Proceedings of the Second Workshop on Management of Replicated Data*, pages 2–5. University of California, Los Angeles, IEEE, Nov. 1992.

[4] 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, Apr. 1992.

[5] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[6] L. Kleinrock. Nomadicity. Presentation at the GloMo PI Meeting (February 4) at the University of California, Los Angeles, 1997.

[7] G. H. Kuenning. *Seer: Predictive File Hoarding for Disconnected Mobile Operation.* PhD thesis, University of California, Los Angeles, Los Angeles, CA, May 1997. Also available as UCLA CSD Technical Report UCLA-CSD-970015.

[8] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 264–275, St. Malo, France, Oct. 1997. ACM.

[9] P. Quéinnec and G. Padiou. Flight plan management in distributed air traffic control system. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, Kawasaki, Japan, Mar. 1993.

[10] D. Ratner, G. J. Popek, and P. Reiher. Peer replication with selective control. Technical Report CSD-960031, University of California, Los Angeles, July 1996.

[11] D. Ratner, G. J. Popek, and P. Reiher. The ward model: A scalable replication architecture for mobility. In *Workshop on Object Replication and Mobile Computing*, Oct. 1996.

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

[13] P. Reiher, T. Page, S. Crocker, J. Cook, and G. Popek. Truffles—a secure service for widespread file sharing. In *Proceedings of the The Privacy and Security Research Group Workshop on Network and Distributed System Security*, Feb. 1993.

[14] P. Reiher, J. Popek, M. Gunter, J. Salomone, and D. Ratner. Peer-to-peer reconciliation based repli-

cation for mobile computers. In *Proceedings of the ECOOP Workshop on Mobility and Replication*, July 1996.

[15] M. Satyanarayanan. The influence of scale on distributed file system design. *IEEE Transactions on Software Engineering*, SE-18(1):1–8, Jan. 1992.

[16] 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, Apr. 1990.

[17] M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kumar, and Q. Lu. Experience with disconnected operation in a mobile computing environment. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pages 11–28, Cambridge, MA, Aug. 1993. USENIX.

[18] 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 Principles*, pages 172–183, Copper Mountain Resort, Colorado, Dec. 1995. ACM.

[19] R. Want, B. N. Schilit, N. I. Adams, R. Gold, K. Petersen, D. Goldberg, J. R. Ellis, and M. Weiser. An overview of the PARCTAB ubiquitous computing experiment. *IEEE Personal Communications Magazine*, 2(6):28–43, Dec. 1995.

David Ratner received a BA in Mathematics and a BA in Computer Science from Cornell University in 1991. He received a MS and PhD in Computer Science from the University of California, Los Angeles, in 1995 and 1998 respectively, where he concentrated on distributed systems. He currently works at Software.Com, Inc. as a development manager and a member of the product architecture group. Dr. Ratner is a member of the IEEE.
E-mail: dratner@excite.com

Peter Reiher is an Adjunct Associate Professor in the Computer Science department at UCLA. Dr. Reiher received his BS from the University of Notre Dame, and his MS and Ph.D. from UCLA. Dr. Reiher's research interests include data replication, active networks, network security, middleware for mobile computing, distributed operating systems, and optimistic methods for parallel discrete event simulation. Dr. Reiher is a member of the ACM.
E-mail: reiher@cs.ucla.edu

Gerald Popek is Chief Technical Officer of NetZero, the fourth largest ISP in the world, as well as being professor of Computer Science at UCLA. Prior to joining NetZero, he served as CTO of CarsDirect.com and of Platinum Technology. He founded Locus Computing Corp, at the time the largest independent developer of Unix systems software. In earlier research on distributed systems, his team created Locus, one of the first full-function and high-performance distributed operating systems. Locus formed the basis for IBM's distributed Unix as well as Tandem's Unix-based solution for telecommunications. Popek's research has focused on computer security, distributed systems, replication, and systems architecture. He served on the Defense Science Board that chose the initial technology for the Internet.
E-mail: popek@cs.ucla.edu

Geoffrey H. Kuenning is an assistant professor of computer science at Harvey Mudd College in Claremont, California. Dr. Kuenning received his BS and MS in computer science from Michigan State University, and his Ph.D. from UCLA. His research interests include file systems, operating systems, mobile computing, distributed systems, active networks, and disconnected operation. Dr. Kuenning has also worked as a software consultant in the fields of operating systems, embedded systems, networking, and graphics. He is a member of ACM, IEEE, Usenix, and CPSR.
E-mail: geoff@cs.hmc.edu