Peer Replication With Selective Control*

David H. Ratner, Peter L. Reiher, Gerald J. Popek, and Richard G. Guy

Computer Science Department
University of California, Los Angeles
{ratner,reiher,popek,rguy}@cs.ucla.edu

Abstract. Many mobile environments require optimistic replication for improved performance and reliability. Peer-to-peer replication strategies provide advantages over traditional client-server models by enabling any-to-any communication. These advantages are especially useful in mobile environments, when communicating with close peers can be cheaper than communicating with a distant server. However, most peer solutions require that all replicas store the entire replication unit. Such strategies are inefficient and expensive, forcing users to store unneeded data and to spend scarce resources maintaining consistency on that data.
We have developed a set of algorithms and controls that implement *selective replication*, the ability to independently replicate individual portions of the large replication unit. We present a description of the algorithms and their implementation, as well as a performance analysis. We argue that these methods permit the practical use of peer optimistic replication.

1 Introduction

Mobile computers frequently experience variable, sporadic, unpredictable, or weak connectivity. Optimistic replication is one well-known method of providing high quality data access in the face of such network conditions [3, 4, 7, 12, 14]. Replication improves performance and reliability by creating multiple copies of important data; optimism allows replicas to be independently updated.

There are two models of optimistic replication. In the *peer-to-peer* (or simply peer) model, all replicas are equals, allowing any replica to communicate with any other. Any-to-any communication is important in mobile situations, when a group of laptops is well-connected within itself but weakly connected or disconnected from distant servers. Additionally, when traveling long distances, often communication with a local partner is easier, more efficient, and more cost effective than synchronization over long-distance links. Finally, any-to-any communication provides advantages for environments where the network topology is variable or unpredictable: replicas are allowed to communicate, exchange updates, and synchronize with whoever is available, rather than waiting for specific servers to become accessible.

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

However, *client-server* models have traditionally provided superior replication control. Whereas all peers typically store the same set of data, clients in the client-server model can store any subset of the server's data. This flexibility permits clients with less power and storage capacity than servers to still replicate data: clients simply select the desired data set and ignore the remainder of the server's data. Clients can therefore replicate very efficiently as they reduce disk space by storing only the relevant data, communication load by only transferring relevant data, and computational load by only maintaining consistency on the data being used.

Our goal is to combine the replication control of the client-server model with the rich information-exchange capability of the peer model. We call our solution *selective replication*. In the volume scenario,[1] we allow individual files from the volume to be replicated independently. Each volume replica stores just the files from the volume that it deems important. Incorporating selective replication control into the peer model required extensive algorithmic modifications, but we discovered that the flexibility could be added with little or no negative impact, and with substantial benefit. Our solution has been implemented in three replication systems, FICUS [7], RUMOR [2], and ROAM [9], and is the basis for hoarding systems such as SEER [5]. The remainder of the paper will discuss the work in the context of ROAM, as that is the most advanced of the three systems.

2 Optimistic Replication

The guiding principle of optimistic replication is that all accessible replicas should be available for full use, including update. Even isolated or partitioned users can generate updates, which is required for mobile computing. However, partitioned updates implies allowing concurrent updates and resulting *conflicts*, so optimistic schemes must reliably detect these conflicts after the fact. Once detected, conflict resolution must occur before normal file activity on conflicted files resumes. Experience with optimistic replication has shown that conflicts are rare and often automatically resolvable [11].

ROAM [9] provides optimistic replication with peer-based, file-level selective replication. Any file within the volume can physically reside at an arbitrary set of the volume replicas. File replicas can be dynamically added or deleted at any time. ROAM is implemented entirely at the user level, and currently runs on Linux and FreeBSD. ROAM uses a multi-level clustering technique capable of providing hundreds to thousands of peer-based replicas.

ROAM maintains data consistency with a periodic synchronization process called *reconciliation*. Reconciliation is a pairwise process between two replicas: a target replica synchronizes with a source by "pulling" the necessary data and status for shared files. When completed, the target knows all information known at the source. Scalability in ROAM is achieved, in part, by "clustering" the reconciliation process.

[1] A volume is smaller than a file system but larger than a directory. For example, a user's home directory and all sub-directories might constitute a volume.

All replicas need not be mutually accessible, but they must form a connected graph. As replicas independently communicate with each other, information is distributed and gossiped to all participants. The particular communication patterns among replicas form the *reconciliation topology*.

Updates are tracked using *version vectors* [8], which is a vector of counters, with one counter per replica. Each counter i tracks the total number of known updates generated by replica $\mathtt{R}_i$. Each replica independently maintains its own version vector for each replicated file; comparing two version vectors compares the update histories of the corresponding file replicas.

Part of maintaining consistency is performing *garbage collection*—the deallocation of resources held by unnamed file system objects [1]. While a relatively simple process in a centralized system, dynamic naming and potentially long-term communication barriers make garbage collection more difficult in partially replicated, distributed systems. Roam uses a fully distributed, two-phase, coordinator-free algorithm to ensure proper distributed garbage collection. Roam additionally interlaces this proven-correct algorithm with data-removing optimizations to free disk space and remove user data in the optimal time possible.

3 Selective Replication Design

Instead of a purely file-granularity design, we maintain the volume as a large-granularity abstraction and permit fine-grain file replication within each volume. Volumes provide several benefits, even if replication is required at a finer granularity, including integrity boundaries, locality and performance benefits, and assistance in naming.

But since we require the flexibility of fine-grain replication within the large-grain volume, we therefore provide the ability to selectively replicate individual files within the volume. *Partial volume replicas* only maintain the data structures and replica information for the selected portion of the volume that they store. Figure 1 depicts a partial volume replica example.

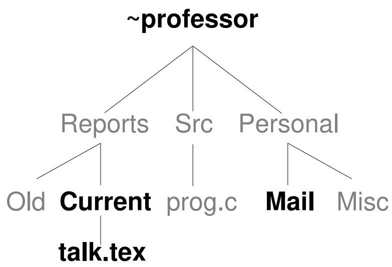

Fig. 1. A partial volume replica. Files listed in bold are stored locally; the others are not.

3.1 Maintaining Replication Information

Sites must determine which files are stored at which replicas to maintain data consistency. Many different solutions are possible. Centralized schemes and replicated schemes based on conservative protocols place undesirable restrictions on when replication factors can change. Therefore, we implemented an optimistic strategy for the maintenance of replication information.

Each replica maintains a *status vector* for each locally stored file. The status vector consists of N elements, where N is the number of volume replicas. Each element is a storage *status value* indicating what information the corresponding volume replica stores about the file.

The two most user-visible status element values (*data* and *nothing*) indicate whether a file's contents are stored at the particular volume replica. Each storage site records its view of where copies of the file reside, and sites that don't store the file (volume replica 2) have no storage overhead. Status vectors are only locally maintained for locally stored files.

Since status vectors are independently and optimistically maintained structures, changes to replication status while partitioned can generate inconsistencies, resulting in *conflicting* status vectors. For example, volume replica 1 could contain the tuple $\{2,$ *data*$\}$ in its status vector while volume replica 3 has the tuple $\{2,$ *nothing*$\}$, resulting in a conflict. For flexibility, we permit any site to order dropping of a replica on any other site, so a site's own status vector is not necessarily correct even for itself.

Conflicting status vectors are resolved by applying a version vector approach to the status vector. Each element is implemented as a monotonically increasing counter, like the version vector. A simple function maps integers to the set of status values; with this approach, two status values can be compared using the standard integer "greater-than" function. Conflicting status vectors can always be automatically resolved by performing a pairwise comparison of status values and selecting the larger one in each case.

3.2 Local Data Availability

The set of all locally stored files at a given volume replica forms a forest of trees. If the forest is permitted to be disconnected, communication with another volume replica would be required to traverse through the intermediate, locally unstored directories that logically connect the disconnected trees. During periods of network partition or disconnection, locally stored data would no longer be available, forcing users to pay close attention to the physical mapping of the tree structure onto the set of volume replicas.

To ensure local availability, we automatically enforce *full backstoring*. Each locally stored file must have its parent directory stored locally as well (the invariant does not apply across volume boundaries). Figure 2 illustrates full backstoring as applied to the partial volume replica from Figure 1.

A common alternate solution employs "prefix pointers" [17], a second directory structure used to connect the user's disconnected namespace. While prefix-pointer solutions avoid storing intermediate directories, they must maintain an

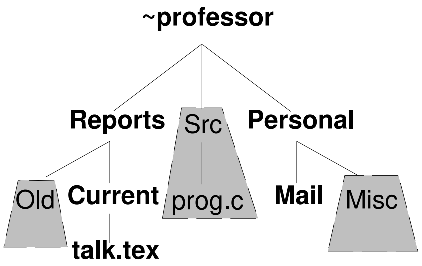

Fig. 2. Full backstoring as applied to the partial volume replica from Figure 1. To store the `talk.tex` and `Mail` files, the intermediate directories must be stored as well. Shaded subtrees are not stored locally.

independent directory structure and integrate it into the user's namespace. Full backstoring avoids the complexity of dual directory structures, and is therefore simpler to implement. Since directories are typically much smaller than files, full backstoring consumes only a small percentage of the available disk space.

Unix hard links pose problems for full backstoring. Given one file name, it is difficult to find all hard links to the same file without scanning all volume replicas. Therefore, we only guarantee the full backstoring of one link; additional paths can be stored at the user's request using the replication tools discussed in Section 3.3. However, the problem is not serious in practice, because hard links are rare, and many non-Unix systems do not even support them. Symbolic links, which are far more common, cause no problems for full backstoring.

3.3 Replication Controls

A volume-based replication service has a single policy for file creation—all files are replicated at all volume replicas. The only necessary controls are those that create, add, or delete volume replicas. Since these actions are relatively rare, simple administratively oriented solutions suffice. Robust selective replication implies multiple policies for file creation. Furthermore, users' desires and data demand patterns change, requiring dynamic addition and deletion of file replicas.

Both novices and sophisticated users need reasonable default replication policies. By default, we store new files wherever the parent directory resides, which has proven a reasonable approach. However, since volume roots are always fully replicated, their children, both directories and files, would be created fully replicated as well. Therefore, we provide user-specified replication *masks* on a per-directory basis, allowing users fine control of replication.

Users may also need to dynamically change the replication factors of individual files; that is, *delete* and *add* file replicas. These actions can occur locally or remotely on another replica's behalf.

Replica deletions should only affect storage locations, not the file's existence or physical data. However, data can be inadvertently lost when the replica being deleted contains the most recent version. To guard against data loss, another replica should synchronize the file with the replica being deleted. *Reverse reconciliation* is not always possible, however, as network partitions can make other replicas temporarily inaccessible. We allow users to bypass reverse reconciliation when necessary, although it is an unsafe operation.

Replica deletion at one site is indistinguishable from new object creation at another. Guy [1] describes a solution for resolving the create/delete ambiguity, which we adopt. We keep a temporary record of the action and executes a distributed algorithm to guarantee both that all replicas learn of the action and that the record is eventually removed. For this purpose, the temporary record is a new state, *dropping*. All data-storing replicas participate in a distributed algorithm that guarantees that all replicas correctly learn of the transition. Full details of this algorithm can be found in [10].

Adding file replicas is simpler than deleting them, because there is no risk of data loss, and there is no create/delete ambiguity (by design, the lack of a delete record implies add). The only additional action is possibly adding extra directories, to enforce full backstoring.

4 Maintenance of Consistency

Data consistency in a selectively replicated environment requires more complete algorithms and stronger communication assumptions than in a fully replicated environment. Without selective replication, a given replica `A` can reconcile all data and learn about all updates by communicating with only a single other site `B`. In a selectively replicated environment, such as depicted in in Figure 3, `A` may store a different set of data from `B`, as shown in the figure. If `A`'s set is not a subset of `B`'s, then `A` can neither reconcile all of its data with `B` nor learn about all new updates from `B`. Reconciliation with other sites is required—site `D` in the case of Figure 3. Forcing `B` to maintain enough information such that `A` could completely synchronize with it would be tantamount to requiring that `B` store a superset of `A`'s data, nullifying most if not all of the advantages of selective replication.

Consistency in this context requires the distribution of updates and the garbage collection of unnameable file system objects.

4.1 Distribution of Updates

Reconciliation maintains data consistency, and the *reconciliation topology* is the communication pattern used between replicas. One replica *pulls* data from another, and in doing so learns all relevant knowledge stored at the source. While a two-way protocol has its advantages, notably performance, a pull-only strategy is more general.

The underlying algorithms are topology-independent: they only require that information can flow from any replica to any other in a finite number of steps and through a finite number of other replicas. However, different topology patterns

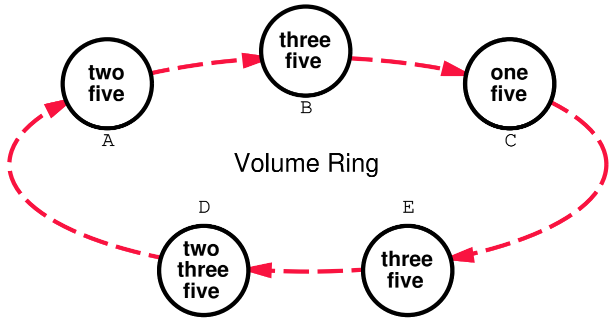

Fig. 3. An example of four selectively replicated files, named by their replication factor. Circles represent volume replicas; arrows indicate data flow. In this topology, reconciliation cannot make progress without forcing the non-data-storing replicas to store file data.

yield different results in terms of performance and message volume complexity [16]. Roam uses a ring topology for propagating updates, with enhancements to handle selective replication. Also, Roam's scalability is based on clustering enhancements to the ring topology [9].

In a ring topology, participants reconcile with the "next" member in the set of volume replicas V, ordered by replica identifier. The ring is dynamically and independently reconfigured by each replica when the set V changes in size. Also, if the preferred reconciliation partner is inaccessible, a node reconciles with the next accessible replica. The clustering enhancements [9] group replicas into local collections to manage the problems caused by mobility, such as "stretching" or "contorting" the ring of replicas. In this way, we provide good performance and efficiency even when mobile replicas effectively destroy the previous ring.

The *adaptive ring* requires only a linear message complexity of $O(|V|)$ messages to propagate information to all replicas. Replicas gossip, transmitting information received about the state of other replicas to those which follow. Thus, the topology does not require point-to-point links inter-connecting all replicas.

Unfortunately, the adaptive ring as described above only operates correctly in a fully-replicated scenario. As seen in Figure 3, selective replication invalidates the implicit assumption that any node can receive and propagate all information via any other node.

A *multi-ring* topology retains the ring's robustness but still correctly supports selective replication. The multi-ring strategy associates an adaptive ring with each individual file. The file ring is also adaptive.

Reconciliation for each file proceeds along that file's own per-file ring, as illustrated in Figure 4. However, we must address the performance implications raised by a multi-ring strategy. In practice, a ring per file is impractical. Roam consolidates files in a volume into *replica sets*, groups of files that are all stored at common volume replicas. Replica set information is managed and maintained in an optimistic fashion, like the data itself, based on replica-local information. As part of reconciliation, the local replica will adjust not only versions of local data but also meta-data, such as where the other replicas are physically stored. Full details on how replica sets are formed and used can be found in [10].

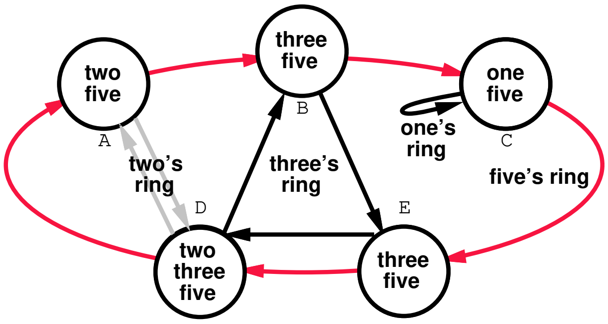

Fig. 4. The same four files as in Figure 3, with each per-file ring indicated. Arrows indicate the data flow between volume rings, labeled by the data transported over them. Together all four rings comprise the multi-ring topology for the given replication pattern.

4.2 Garbage Collection

Garbage collection deallocates file system resources (disk space, inodes, etc.) after file removal. Actual deallocation is straightforward, but determining the correct time to do so is difficult. Dynamic naming[2] and long-term communication delays imply that files might temporarily have no local links, but still be globally accessible.

5 Performance Evaluation

This section describes the measurements, experiments, and benchmarks used to evaluate selective replication, as well as the results and conclusions that can be drawn from them. We begin with a description of the data overhead imposed by selective replication, describe the performance of reconciliation, and conclude with some comments on real use and future evaluation work.

The measurements and benchmarks were performed on Roam.

[2] Files in systems like Unix can have multiple names created dynamically, either by link or rename operations.

5.1 Data Overhead

Selective replication requires adding a status vector to the data structures already stored for full replication. Each replica in the *data*, *attributes-only*, or *dropping* state stores a status vector consuming 4N bytes, where N is the number of volume replicas. Replicas in the *nothing* state store no attributes at all.

We studied the effect of selective replication on the disk overhead using five different replication patterns. The volume used is a 13.6MB volume, consisting of 307 files, of which 38 are directories. It contains large files (up to 3.5MB) and small files (a few bytes), large directories (the volume root contains 151 files) and small directories. Using the selective replication tools, we created five different replication "patterns," summarized in Table 1. For ease of recognition, we have given each pattern a simple, mnemonic name.

Pattern	Objects	Directories	Size
small	173	38	2.6MB
space-saver	295	38	5.75MB
inode-saver	218	38	11.21MB
subtree	256	31	12.9MB
full	307	38	13.6MB

Table 1. The five replication storage patterns used for studying the effects of selective replication. Each pattern has a different subset of the 13.6MB volume. The complete volume stores 307 files, of which 38 are directories. In each case, the total number of file system objects *includes* the number of directories.

We created two replicas, performing the selective replication changes and measuring the results at the second replica. Replica 1 stored a fully replicated volume; replica 2 stored different subsets of the volume, according to the five replication patterns. Figure 5 illustrates the results, showing two different curves. The first curve indicates the percentage of disk overhead as a function of the amount of locally stored user data, which is one measure of the cost of selective replication. The second shows the savings in disk space provided by selective replication (including the overhead), which is one benefit of selective replication.

Replication pattern "small" has the smallest average file size; we therefore expect it to have the highest percentage of overhead. As the average file size gets larger, the percentage of overhead becomes smaller.

The second set of bars in the graph illustrates a benefit of selective replication. We preserve access to all files in the volume, while potentially saving the user up to 80% of the required disk space, as in the case of pattern "small."

5.2 Synchronization Performance

The key performance figure for Roam is the actual performance of synchronization under different selective replication patterns and workloads. We performed

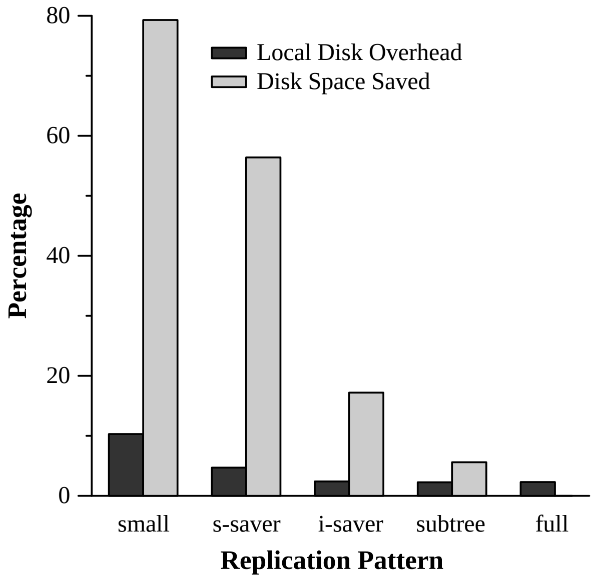

Fig. 5. Selective replication disk overhead for the 13.6MB volume. The replication patterns are described in Table 1.

our experiments with two portable machines connected by a 10Mb Ethernet and with minimum daemons running. The first machine is a Dell Latitude XP with a 486DX4 running at 100Mhz and 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. In other words, the reconciliation process always executed on the TI machine.

We utilized the previous five replication patterns at the local (TI) machine. Additionally, since reconciliation performance depends heavily on the sizes of the files that have been updated, we varied the amount of data updated at the remote site (and therefore the amount to be transfered by reconciliation to the local site) from 0 to 50%. Data files are selected at random for update; they are not predetermined. We performed seven runs for each data point; the results are illustrated in Figure 6.

Selective replication saves the user up to half of the cost of replication. The amount of data updated at the remote side in pattern “small” makes little difference, since almost none of it is replicated at the local replica. In contrast, for the “full” replication pattern, reconciliation time rises by almost one-third when the amount of data being modified increased from 0 to 50%. The figure demonstrates a second clear benefit to selective replication, in addition to the disk space savings demonstrated above: reconciliation takes less time.

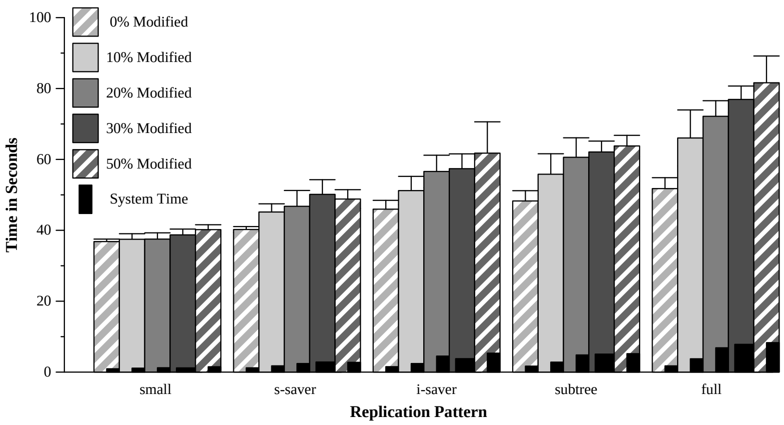

Fig. 6. Reconciliation performance for the 13.6MB volume. The replication patterns are described in Table 1. 95% confidence intervals are shown. Except where indicated, all measurements are of elapsed time.

Even when nothing was modified at the remote site, reconciliation took longer under replication pattern "full" than replication pattern "small." Since more files are stored locally in replication pattern "full," reconciliation spends more time detecting if any updates have been generated locally (to update its local data structures and version vectors), regardless of whether or not updates were applied at the remote site.

Selective replication has been used in production with Ficus, Rumor, and Roam. Usually, use was confined to a static model: users dropped specific files or sub-trees from specific replicas, and didn't again change the replication patterns. Typically, such uses were motivated by space and/or time restrictions. Additionally, files that didn't need to be stored at certain replicas were dropped from them, to reduce reconciliation time.

6 Related Work

Coda [4, 12] is an optimistic replicated file system constructed on the client-server model, as opposed to the peer architecture. Coda trivially provides selective-replication control at the clients, but not at the replicated servers which are, by definition, peers. Additionally, the clients cannot inter-communicate due to the inherent restrictions of the client-server model. The restrictions dramatically simplify the algorithms needed to manage consistency, at the cost of limiting the system's utility for mobile workgroups and other environments.

Coda has greatly optimized the communications and synchronization between client and server, especially in environments with weak connectivity [4, 6]. Additionally, Coda uses a vastly simplified garbage collection strategy at their replicated servers, based on logging directory operations. The viability of

CODA's solution depends on how often the logs become full, as dropping some log information can lead to unnecessary conflicts. Log wrap-around could result from long disconnections or a plethora of file system activity. Should the CODA strategy be applied in a mobile environment, where disconnections are commonplace, the log wrap-around strategy could result in a large number of unnecessary conflicts.

The BAYOU system [14] is also a replicated storage system. Like FICUS and RUMOR, BAYOU is based on the peer-to-peer architecture. However, BAYOU does not yet support any form of selective replication.

The DECEIT file system [13], places all files into one "volume" and allows each individual file to be replicated independently with varying numbers of replicas. However, DECEIT employs a conservative writer-token approach to replication, and thus cannot provide the high availability offered by optimistic mechanisms. Also, DECEIT cannot tolerate long-term network partitions.

The university version of the LOCUS operating system [15] provides volumes and allows selective replication within the volume. However, the approach taken toward replication is not strictly optimistic.

7 Conclusions

Peer models are necessary in a wide variety of environments, but have traditionally provided less flexibility than client-server models. Selective replication provides the replication flexibility of the client-server model while maintaining the advantages of a peer architecture. By enabling any-to-any communication and providing the ability to store dynamic subsets of replication units, the functionality and flexibility of the two models have effectively been merged.

Even without selective control, peer replication is important for the rich information model it provides. However, there are many environments where peer solutions are simply unusable in practice without selective replication due to the inherent costs of traditional peer models. Selective replication allows peer solutions to be effectively and efficiently utilized in scenarios that otherwise would prove impractical. With the rapid rise of mobility and mobile workgroups, peer solutions will be required even more than before, making selective replication an integral part of any replication service designed for a mobile world.

References

1. R. G. Guy, G. J. Popek, and T. W. Page, Jr. Consistency algorithms for optimistic replication. In *Proceedings of the First IEEE International Conference on Network Protocols*. IEEE, Oct. 1993.
2. R. G. Guy, P. L. Reiher, D. H. Ratner, M. F. Gunter, W. Ma, and G. J. Popek. *Rumor*: Mobile data access through optimistic peer-to-peer replication. In *Advances in Database Technologies:ER '98 Workshops on Data Warehousing and Data Mining, Mobile Data Access, and Collaborative Work Support and Spatio-Temporal Data ManagementProceedings of the*, number 1552 in Lecture Notes in Computer Science, pages 254–265. Springer Verlag, Nov. 1998.

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. J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.
5. 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.
6. L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 143–155, Copper Mountain Resort, Colorado, Dec. 1995. ACM.
7. T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software—Practice and Experience*, 28(2):155–180, Feb. 1998.
8. D. S. Parker, Jr., G. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.
9. D. Ratner, P. Reiher, and G. J. Popek. Roam: A scalable replication system for mobile computing. In *MDDS'99: 2nd International Workshop on Mobility In Databases and Distributed Systems @ DEXA'99*, Sept. 1999.
10. D. H. Ratner. Selective replication: Fine-grain control of replicated files. Master's thesis, University of California, Los Angeles, Mar. 1995. Available as UCLA technical report CSD-950007.
11. 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, Boston, MA, June 1994. USENIX.
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, Apr. 1990.
13. A. Siegel, K. Birman, and K. Marzullo. Deceit: A flexible distributed file system. In *USENIX Conference Proceedings*, pages 51–61. USENIX, June 1990.
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 Principles*, pages 172–183, Copper Mountain Resort, Colorado, Dec. 1995. ACM.
15. B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 49–70. ACM, Oct. 1983.
16. A.-I. Wang, P. Reiher, and R. Bagrodia. A simulation evaluation of optimistic replicated filing in mobile environments. In *Proceedings of the 18thProceedings of the 18th IEEE International Performance, Computing, and Communications Conference*, Feb. 1999.
17. B. Welch and J. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed system. *Sixth International Conference on Distributed Computing Systems*, pages 184–189, May 19-23, 1986.