Peer Replication with Selective Control*

David Ratner Gerald J. Popek† Peter Reiher
Department of Computer Science
University of California, Los Angeles

Abstract

Peer-to-peer optimistic replication strategies provide improved functionality over traditional client-server models by enabling any-to-any communication. This is especially useful in environments with common network partitions, such as mobile environments, in which communicating with a closely connected peer can be cheaper than communicating with a distant, weakly connected server. However, most peer solutions require full replication, meaning that all replicas store the entire replication unit, whether it be a volume or a database. Such strategies are clearly inefficient and not cost effective. Users are forced to both store more data than they require and spend time and money maintaining consistency on that data, potentially making peer solutions unusable in practice.

We have developed a set of algorithms that implement selective replication, allowing individual portions of the replication unit to be replicated independently. We have implemented our solution in Ficus and allow files from the volume to be selectively replicated without storing the entire volume. We present a description of the algorithms and their implementation in the Ficus file system, as well as a performance analysis. We argue that these methods permit the practical use of peer optimistic replication.

1 Introduction

The need for optimistic replication in environments with variable, sporadic, unpredictable, or weak connectivity is already well known [4, 6, 7, 9, 20, 24]. Replication provides both improved performance and reliability through the creation of multiple replicas of important data objects; optimism allows replicas to be independently updated, and guarantees that concurrent updates will be resolved when they are detected.

There are essentially two different models of optimistic replication. One has traditionally provided a richer informational model, while the other has historically offered greater replication control. In the *peer-to-peer* (or simply peer) model, all replicas are equals. Superior functionality is provided by allowing any replica to communicate with any other. Any-to-any communication becomes extremely important in mobile situations, when two (or more) laptops find themselves well-connected while being only weakly connected or entirely disconnected from distant servers. Such usage is often referred to as *mobile workgroups*. Additionally, the ability for any-to-any communication provides advantages for other environments where the network topology is often variable or unpredictable: replicas are allowed to communicate, exchange updates, and synchronize with whomever is available, rather than waiting for specific servers to become accessible.

However, *client-server* implementations 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. The flexibility permits clients that are physically less powerful and have less storage capacity than the servers to replicate data: they simply select the desired data set and ignore the remainder of the server's data. Clients can therefore replicate much more 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 chose to add improved replication control to the peer model for many reasons. First, peer functionality is important, evidenced by the rise of mobile use. Mobility by its very nature requires peer capabilities; for example, mobile workgroups. A number of research projects are investigating peer usage, such as xFS in the NOW project [1] and the Bayou system [24]. Second, we expected that

*This work was sponsored by the Advanced Research Projects Agency under contract N00174-91-C-0107.

The authors can be reached at the Department of Computer Science, UCLA, Los Angeles, CA 90095, or by email to {ratner,popek,reiher}@cs.ucla.edu.

†Gerald Popek is also affiliated with Platinum *technology*.

modifying a fundamentally client-server design to incorporate a rich communications structure would require such basic structural changes as to imply effectively starting from scratch. Adding selective control to a peer model ought to better preserve the underlying framework.

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; additionally, this set can be dynamically modified with simple tools. Incorporating this control into the peer model required extensive algorithmic modifications, which will be explained in the Ficus [4, 14] context. Finally, we discovered that the flexibility could be added with little or no performance impact.

The next section provides a brief overview of Ficus and optimistic peer replication. Section 3 describes the overall selective replication design; data consistency maintenance is discussed immediately following. Section 5 describes our performance measurements. Related work is discussed in Section 6, followed by our conclusions.

2 Optimistic Replication and Ficus

The guiding principle of optimistic replication is that all accessible replicas should be available for full use, including update. Optimistic replication therefore provides a highly available filing service, because even isolated or partitioned users can generate updates—a capability required for mobile computing. However, allowing partitioned updates provides the potential for 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.

Ficus [4, 14] is one system that supports optimistic replication. With a peer-to-peer model that allows any two replicas to synchronize with each other, and with the work reported in this paper, Ficus supports peer-based, file-level selective replication. Each file can reside at an arbitrary number of volume fragments. File replicas, like the volume replicas that contain them, can be dynamically added or deleted at any time. Additionally, Ficus provides transparent access to remote replicas; users need not store a particular file or volume in order to access it.

Ficus maintains data consistency with two separate mechanisms. At update time, *update notification* messages are sent to all accessible replicas in a "one-time, best-effort" manner. A second process called *reconciliation* runs periodically to guarantee consistency in the face of lost or undeliverable update notification messages.

Reconciliation is a pairwise process between two replicas: one replica synchronizes with respect to another by "pulling" the necessary data and status for shared files. 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 (such as a ring or a star) form the *reconciliation topology*. While the reconciliation algorithms are topology independent, the actual topology can affect both the number of messages exchanged between all replicas and the time required to reach resolution.

Updates are tracked using *version vectors* [15]. A version vector is a vector of counters, with one counter per replica. Each counter i tracks the total number of known updates generated by replica i. Each replica independently maintains its own version vector for each replicated file; by comparing two version vectors, the update histories of the corresponding file replicas can be compared.

As part of reconciliation, Ficus contains a distributed algorithm to perform *garbage collection*—the deallocation of resources held by unnamed file system objects [5]. 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. Ficus uses a fully distributed, two-phase, coordinator-free algorithm to ensure that all replicas are knowledgeable about the garbage collection process and can eventually complete it, although any set of participants may never be present simultaneously [5].

Ficus is implemented in SunOS 4.1.1, and supports the research of approximately a dozen people. We have over 256 man-months experience with the system and selective replication control. All Ficus development occurs under Ficus.

3 Selective Replication Design

This section describes the overall selective replication design. Several important design characteristics are worth noting at the outset:

- We manage both files and file groups (volumes) for several reasons described below
- Replication control information is itself managed

[1] A volume is smaller than a file system but larger than a directory [19]. For example, a user's home directory and all subdirectories might constitute a volume.

optimistically, resulting in considerable simplifications

- Locally stored data is always available, even during complete network disconnection
- Files which are not stored locally can be transparently accessed remotely

Each of these items will be addressed below, including motivation, design impact, and implementation effects. The section concludes with a description of the replication policies and tools. Performance considerations are deferred until Section 5.

3.1 A two-level granularity structure

Instead of a file-granularity design, Ficus maintains the volume as a large-granularity abstraction and permits fine-grain file replication within each volume. Volumes provide several key benefits:

Locality and Performance: Logically connected files can be grouped together in one physical place. Performance-intensive tasks are initiated once for the volume, rather than once per volume member.

Integrity: Volumes provide natural firewalls for preventing the propagation of errors and for establishing fundamental security barriers.

Naming: A collection of logically connected files can easily be identified and acted upon as a single unit.

However, as previously mentioned, we require the flexibility of fine-grain replication with respect to 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 which they store.[2] Figure 1 depicts a partial volume replica example.

The design yields a two-level granularity architecture which we believe yields efficient results. The design allows for system functionality to be logically separated by granularity: those that should operate on a large-granularity object for performance reasons, and those that need to be finely controllable for flexibility requirements. For example, security perimeters are more easily and cleanly erected around a volume as a whole, because the volume has a single entry point. Additionally, reconciliation is fairly heavyweight, and performance dictates that it should operate on a large-granularity object instead of being invoked once per file. Of course, the actual replication mechanisms must be fine-grain to provide the flexibility users require, and therefore operate on individual files.

[2] The volume root must always be stored locally.

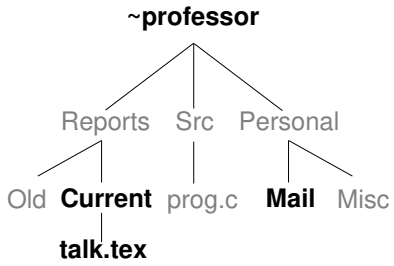

Figure 1: A partial volume replica. Files listed in bold are stored locally; the others are not.

3.2 Maintaining replication information

Sites need to determine which files are stored at which replicas to maintain data consistency and perform replica selection. Many different solutions are possible; for example, central "managers" could record the storage locations for all files within each volume. However, centralized schemes and replicated schemes based on conservative protocols (like primary site) place restrictions on when replication factors can change. For the same reasons that optimism is required for updates to data, optimism is also required for updates to metadata, including physical locations of objects. Therefore we implemented an optimistic strategy for keeping replication information at each replica. Like all optimistic strategies, this approach operates smoothly in the common cases and resorts to special actions in the face of failures—i.e., when the replication information is incorrect.

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. Thus, the basic status vector structure is an array of length N, with each array element being a tuple of the form {`volume replica identifier, status value`}.

A particular status element can have one of several values. The two most user-visible values indicate whether or not file contents are stored at the particular volume replica, referred to as the *data* and *nothing* status values, respectively.[3] For example, the status

[3] Additional status values will be discussed in the following

Replica	Status Vector
1	{(1, *data*), (2, *nothing*), (3, *data*)}
2	none stored
3	{(1, *data*), (2, *nothing*), (3, *data*)}

Table 1: A status vector example. Of the three volume replicas, the file in question is replicated at only two of them (replicas 1 and 3). The status vector is shown as an array of tuples {volume replica identifier, status value}.

vectors for a given file at each of three volume replicas are illustrated in Table 1. 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, partitioned updates (i.e., changes in who stores what) 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*}. Additional mechanism is needed to identify the true status value for replica 2. Not even replica 2 can resolve the conflict, since to permit users dynamic and free decisions regarding where to store files, we allow any replica X to modify any other replica Y's status vector.

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.

3.3 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 directories, not stored locally, 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.

sections.

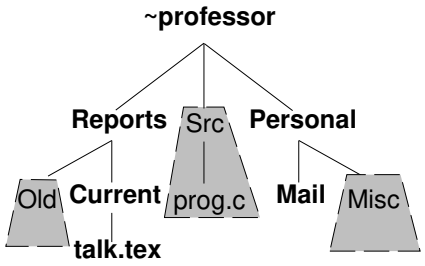

Figure 2: Full backstoring as applied to the partial volume replica from Figure 1. To store the `talk.tex` and `Mail` files, the intermediate directories must be stored as well. Shaded subtrees are not stored locally.

To ensure local availability, Ficus automatically enforces *full backstoring*. This invariant requires that each locally stored file 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. To store the invited talk (`talk.tex`) and the week's mail (`Mail`), the professor must store the intermediate directories (`Reports`, `Current`, `Personal`). However, there is no need for any files under `Src`, `Misc`, or `Old`, so these subtrees are not stored locally.

A common alternate solution employs "prefix pointers" [26], a second directory structure used to connect the user's disconnected namespace. While prefix-pointer solutions avoid storing intermediate directories, they must maintain an 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, the Ficus system only guarantees the full backstoring of one link; additional paths can be fully-backstored at the user's request using the replication tools discussed in Section 3.5. However, this problem is not serious in practice, because hard links are rare, and many modern systems do not even support them. Symbolic links, which are far more common, cause no problems for full backstoring.

3.4 Remote data availability

Selective replication allows individual files to be stored at any physical volume replica; specifically, files may be stored at fewer locations than those of their parent directory. Users who only replicate a particular subtree (or subtrees) locally store only the directory entries corresponding to that subtree (or subtrees), such as in Figure 2. However, continuing with the example, the given replica does not know the physical location of the other subtrees, such as the `Src` subtree. We believe it is essential that access to remotely stored data be transparent when network connections exist; selective replication should maintain name transparency and provide reasonable remote-access performance.

Since the set of volume replicas could be quite large, polling on each remote file access would perform poorly. Instead, an additional status value is defined to locally maintain just the status vector for a particular file without storing file data. Since the status vector indicates which volume replicas store file data, it provides a fast and easy method of locating file replicas; additionally, the status vector is itself small, so it imposes little storage overhead. The new status value is referred to as *attributes-only*.

The system maintains the invariant that, for all locally stored directories, all directory entries are also stored locally in at least the *attributes-only* state (they could be stored in a *data* state). Thus, in Figure 2, `Src`, `Misc`, and `Old` would be maintained locally in an *attributes-only* state. Nothing would be stored for the file `prog.c`; it does not need to be maintained in an *attributes-only* state. Storing the status vector for `Src` is enough to locate a replica of the subdirectory, and therefore a replica of `prog.c`.

The status vector is used only as a hint to the physical storage locations. Since it is optimistically and independently maintained, partitioned updates can create temporary inconsistencies. We could either attempt to maintain consistency on the *attribute-only* replicas or treat them like a cache. Since consistency would be maintained optimistically, there are still scenarios where the *attribute-only* replicas are inconsistent; we therefore treat the replicas like a cache, and resort to a multi-level polling and cache-replacement solution if the cache is invalid [17]. Treating the *attribute-only* replicas like a cache means that there is zero overhead or performance impact on reconciliation.

3.5 Replication controls

A volume-based replication service has a single policy with regard to file creation—all files are replicated at all volume replicas, known as *full replication*. As such, the only necessary controls are those that create, add, or delete volume replicas. Since these actions are relatively rare, simple administratively oriented solutions suffice. In contrast, robust selective replication implies multiple policies regarding file creation. Furthermore, users' desires and data demand patterns change, implying the necessity for dynamic addition and deletion of file replicas. This section discusses replication policies as well as the selective replication utilities.

3.5.1 Replication policies

Reasonable default replication policies are necessary both for novices and more sophisticated users. By default, one could store new files wherever the parent directory resides. This approach has proven a reasonable one in practice. However, since volume roots are always fully replicated, their children, both directories and files, would be created fully replicated as well. In the absence of further mechanism, everything would be fully replicated. Therefore, user-specified replication *masks*, or mappings onto the set of volume replicas, are provided. On a per-directory basis, users can specify where the directory's children should be stored, constrained only by the full backstoring invariant.

3.5.2 Dynamic replica deletion

Users may also need to dynamically change the replication factors of individual files; that is, *delete* and *add* file replicas.[4] These actions can occur locally or remotely on another replica's behalf.

There are two basic deletion issues, the potential loss of updates and the create/delete ambiguity [2]. Both of these are examined below.

Potential loss of updates: 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. In general, it is not possible to determine which replica has the most recent data, given only local information.

To guard against data loss, another replica should reconcile the file with the replica being deleted. Called a *reverse reconciliation*, this process is not always possible, however, as network partitions can make other replicas temporarily inaccessible. Nevertheless, it is sometimes necessary to allow replica deletions in these

[4] File in this context refers to all file system objects, including directories and symbolic links.

cases, despite the potential loss of updates. When disks become full, all work ceases, and often the possibility of losing an update is less important than the certainty of not being able to continue working. We therefore allow users to bypass reverse reconciliation.

Create/delete ambiguity: In the absence of additional information, object deletion (i.e., replica deletion) at one site is indistinguishable from object creation at another.

Guy [5] describes a solution for resolving the create/delete ambiguity in the garbage collection of distributed file system objects. We adopt Guy's solution here. Guy keeps 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. In our case, the record of the action is a new status, in addition to the three already mentioned, called the *dropping* state. When one or more file replicas change state to *dropping*, all data-storing replicas participate in a distributed algorithm. The *dropping-notification* algorithm guarantees that all replicas correctly learn of the transition by one or more replicas into the *dropping* state. The algorithm also guarantees that, after everyone has learned of the transition, the state changes from *dropping* to *nothing*. When sites change their local state to *nothing*, they remove the entire status vector, thus cleanly completing the replica deletion.

3.5.3 Dynamic replica addition

Adding file replicas is simpler than deleting them, because there is no risk of data loss, and there is no create/delete ambiguity to resolve (by design, the lack of a delete record implies add). The full backstoring invariant (Section 3.3) must be maintained, however, as otherwise the new data could be inaccessible during network partitions. Therefore all intermediate directories on the path between the volume root and the to-be-added file must be added as well. Fortunately, the addition of the intermediate directories is relatively simple, because they must be accessible at replica addition time; if they were not, the file itself would not be nameable.

4 Maintenance of Consistency

Most of the selective replication design has already been described, except for the maintenance of data consistency. Data consistency in a selectively replicated environment requires both 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. However, in a selectively replicated environment, A may store a different set of data from B; 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, D in the case of Figure 3. Forcing B to maintain enough information such that A could completely reconcile with it would be tantamount to requiring that B store a superset of A's data—a requirement we do not want to enforce in general.

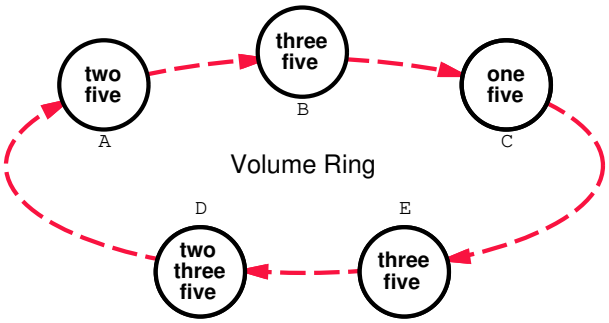

Figure 3: An example of four selectively replicated files, named by their replication factor. Circles represent volume replicas; arrows indicate data flow. In this topology, reconciliation cannot make progress without forcing the non-data-storing replicas to store file data.

This section describes the algorithms and topologies used to maintain consistency in the selectively replicated environment. Consistency in this context incorporates two distinct pieces: the distribution of updates and the garbage collection of unnameable file system objects. Each is described separably.

4.1 Distribution of updates

Reconciliation is the process that maintains data consistency, and the *reconciliation topology* is the communication pattern used between replicas. The underlying algorithms in Ficus 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 yield different results in terms of performance and message volume complexity. For example, quadratic message complexity results from an all-pairs reconciliation topology, but a ring (using a gossip-based transfer of information) reduces the mes-

sage complexity to a linear cost. A superior messaging plan avoids the quadratic cost when inter-site communication is available, but gracefully handles degraded communication.

We will first discuss ring-based topologies and enumerate their advantages. We will then describe how we modify the basic ring for selectively-replicated environments, resulting in a *multi-ring* topology. Implementation and performance aspects of the new topology will be discussed, as well as further optimization opportunities.

4.1.1 Ring-based topologies

We first define a ring-based topology. We identify the set of volume replicas as the set V. In a standard ring topology, participants reconcile with the "next" member in V, ordering V by replica identifier; that is, each replica reconciles with the site corresponding to the next smallest replica identifier. Additionally, we make the ring *adaptive* in two ways:

1. The ring is dynamically reconfigured when the set V increases or decreases in size.

2. During periods of network failures, a given replica $v \in V$ reconciles not with the next replica $v-1$ mod V but with the next *accessible* replica $v-e$ mod V, where $e \in 1, ..., |V-1|$.

The adaptive ring requires only a linear message complexity of $O(|V|)$ messages to propagate information to all replicas. As illustrated by the second adaptation above, the ring adapts itself to changing network topologies and is therefore quite robust. Replicas gossip about the status of other replicas: that is, a given replica transmits information it has received about the state of other replicas to those which follow in the sequence. Thus, the topology does not require point-to-point links inter-connecting all replicas. The adaptive ring allows rarely or never-communicating sites (such as laptops) to share data by relying on third-party replicas to gossip on their behalf. For these reasons, it is an attractive topology to consider.

4.1.2 A multi-ring topology

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. Since files are only stored at specific volume replica subsets, only those subsets can be knowledgeable about the files' state. Forcing other volume replicas to maintain data solely for gossip purposes negates two key advantages of selective replication: saving disk space and limiting communication cost.

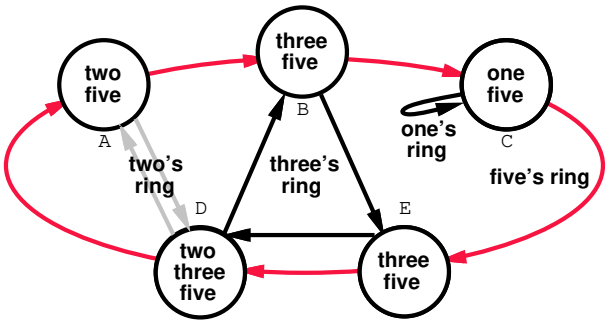

Figure 4: The same four files as in Figure 3, with each per-file ring indicated. Arrows indicate the data flow between volume rings, labeled by the data transported over them. Together all four rings comprise the multi-ring topology for the given replication pattern.

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. By definition, this ring must be a subset of the volume-replica ring, since the file only exists within the volume. The file ring is also adaptive: it responds to replica additions and deletions, and adapts to changing network topologies just as the simple solution did.

Reconciliation for each file proceeds along that file's own per-file ring. However, since reconciliation is fairly heavyweight, it is impractical to initiate a separate reconciliation process per file in each volume. Therefore reconciliation remains a volume-based process, and multiple per-file rings are utilized by each volume replica. Figure 4 illustrates the per-file rings and how reconciliation utilizes them.

4.1.3 Multi-ring implementation

A naive implementation of the multi-ring topology would perform poorly, because n files require n separate but possibly similar per-file rings. If m of those files are stored at similar subsets of replicas, their per-file rings greatly overlap, and the files could, for the most part, be reconciled in a single communication action. This saves the overhead of establishing and closing m communication links and condenses m messages to one. Locality of reference suggests that files stored together

at a single replica are somehow logically related, and therefore other replicas that store one of them are likely to store others as well. This offers hope that m will, in general, be a sizable proportion of n.

We take advantage of locality of reference by reconciling files in *groups*. A group is a property of two communicating volume replicas. It is defined to be the set of files that is mutually stored at both replicas, and takes advantage of duplication or overlap in individual file rings. For example, in Figure 4, the group shared between replicas B and C ($\text{group}_{B,C}$) is {`five`}, $\text{group}_{B,D}$ = {`three, five`}, and $\text{group}_{B,E}$ = $\text{group}_{C,E}$ = {`five`}. Group formation is a straightforward process, performed entirely with local information (the status vector). Of course, communicating replicas might disagree about group membership, since the status vector is optimistically maintained. Disagreements are resolved in the reconciliation details described below.

The group concept becomes the driving force behind the multi-ring topology implementation. Communicating volume replicas talk only in groups, not individual files, which allows all m files from above to be reconciled in a single communication action.

While files will be members of multiple groups as different pairwise combinations of volume replicas are examined, the overall reconciliation topology must still be preserved to ensure correctness. Adherence to the multi-ring topology guarantees that all replicas learn about all updates, and that no replica is accidently "isolated." If B decided to reconcile all its files directly with E in Figure 4, C would be isolated. Blending the multi-ring topology with the group abstraction yields the *first-group* concept, which guarantees reconciliation correctness.

From a volume replica's viewpoint, the first-group for each file is the group shared with the next accessible site in that file's ring. Each file at each volume replica has a specific first-group, corresponding to following the file's per-file ring. Thus, ensuring that each file be reconciled as a member of its first-group guarantees adherence to the multi-ring topology and therefore reconciliation correctness. For instance, in the case of replica B from Figure 4, the first-group for `five` is $\text{group}_{B,C}$ and the first-group for `three` is $\text{group}_{B,E}$. Of course, since the multi-ring topology is adaptive, the first-group definition is as well, so if replica C were temporarily inaccessible, `five`'s first-group would become $\text{group}_{B,E}$. The first-group is easily and optimistically calculated using local information (the status vector). Re-reconciling files as a member of other groups is not required for correctness, though doing so may propagate information more quickly [17].

4.1.4 Reconciliation details

One replica *pulls* data from another, and in doing so learns all relevant knowledge stored at the source. A pull-based strategy is more general than a two-way communication model and provides support for predominantly one-way transport mechanisms such as floppy disks.

When one replica pulls a group of files from another, it detects and resolves possible discrepancies in group membership (caused by incomplete local information). The discrepancies (discussed more fully in [17]) could be caused by:

- New file replicas (file creation or remote file replica addition) expected to be stored at the other replica
- Files which have completed one of the two create/delete resolution algorithms (garbage collection and dropping-notification), and therefore no longer exist at one of the replicas

Distinguishing the different scenarios is straightforward, and the appropriate action is taken in each case. Additionally, the status vectors for each file need to be compared to gossip about status changes. The comparison is performed using the method described in Section 3.2. Finally, the "standard" reconciliation actions must be performed, such as version-vector comparisons, update propagation, and conflict-detection, discussed in [4, 14, 18].

4.1.5 Optimization opportunities

The multi-ring topology is clearly not ideal in all environments. A single ring is formed with all replicas as members, instead of, for example, utilizing the natural clusters formed by geographic location. We are currently analyzing more sophisticated topologies.

4.2 Garbage collection

Garbage collection deallocates file system resources after file removal. Resources such as disk space occupied by file data must be returned to the general resource pool. Actual deallocation is straightforward, but determining the correct time to do so is difficult. Dynamic naming[5] and long-term communication delays imply that files might temporarily have no local links, but still be globally accessible.

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

We first describe characteristics of the garbage collection problem in a distributed, replicated environment, followed by an overview of the solution we employ. We conclude the section with a few remarks about optimizations and performance improvements.

4.2.1 Garbage collection issues

Performing garbage collection in a *fully replicated* environment is itself a complex problem. First and foremost, one must guard against premature garbage collection. Garbage collecting before all participants realize the process has started generates create/delete ambiguities [2, 5] between the informed and uninformed replicas. Nevertheless, since the data being garbage collected occupies otherwise usable disk space, every effort should be made to garbage collect quickly.

Additionally, files may temporarily have no local links while still being globally accessible: new links could exist at other replicas, waiting to be propagated from informed to uninformed sites. The Ficus *no lost updates* semantics [4] require that the latest data version must be preserved as long as the data is globally accessible. Therefore, consensus that no links exist must be reached globally before garbage collection can properly occur.

Selective replication only adds more complexity. In the case of full replication, the garbage collection participants are well-known prior to algorithm initiation; that is, all volume replicas participate. In contrast, selection replication allows a dynamic subset of the volume replicas to physically store any particular file. Since the replication control information itself must be managed optimistically to maintain high availability, sites cannot be certain they know the complete and correct set of participants upon garbage collection initiation.[6] Furthermore, to permit free and dynamic decisions regarding where to store files, remote replica additions and deletions are allowed; that is, operations relevant to a particular replica can be executed remotely and subsequently propagated. Such flexibility, combined with the race condition between the distribution of a creation action and its impending deletion, implies that at garbage collection initiation, a given replica might itself be uninformed with regard to its own participation in the distributed algorithm.

4.2.2 Garbage collection solution

We utilize a two-phase, coordinator-free, distributed solution to the problem of garbage collecting in a replicated graph [5], and augment the solution for selective replication. Basically, the algorithm is a distributed implementation of a two-phase commit protocol among multiple participants, and verifies that no member completes before all members know that completion is imminent. Any site with no local links for a file initiates garbage collection; the algorithm either stops when a new link is discovered or reaches consensus that no links exist globally, completing garbage collection.

The solution in [5] assumes full replication. To support selective replication, the solution must be substantially extended to execute only between data-storing replicas of the file. As explained above, the set of data-storing replicas is dynamic and not necessarily known upon garbage collection initiation. Therefore, while forming consensus on the number of global links, we also form consensus on the set of participants, so that upon normal termination, all participants know the correct set.

Forming consensus on the participant set can be difficult. Distributing information in this selectively replicated environment employs the per-file ring; however, file removals (which initiate garbage collection) remove the only marker—the file name or directory entry—that provides for ring construction. A race condition therefore exists between the construction of the per-file ring and the distribution of the deletion-action.

In cases where the ring never fully forms prior to its destruction, some sites might be unaware that they are in fact participants. Their non-participation would block the algorithm from ever completing at any site. Our solution involves having the knowledgeable sites *push* the data to the unknowledgeable ones when required. This *push-pull* mechanism guarantees that the participant set correctly forms and allows consensus on set membership to be reached. Utilizing a push for information transferal is only feasible when sites can be classified as knowledgeable or unknowledgeable. Pull-style mechanisms do not require this additional complexity, and as such are generally simpler to design and build. However, since garbage collection itself requires verifying that each participant has performed a specific action, the classification already exists, and the push mechanism can be easily implemented. Specific details on building the participant set, reaching consensus, and pushing data between replicas can be found in [17].

5 Performance Evaluation

This section describes the measurements, experiments, and benchmarks used to evaluate selective replication, as well as the results and conclusions which can be drawn from them. We begin with a description of

[6] This is different from dynamic creation and deletion of new volume replicas, which can be trivially handled.

the data overhead imposed by selective replication, describe the benchmarks and performance data, and conclude with some future work. Some of the comparisons are performed against an earlier, volume-only replication version of Ficus, which we call VFicus. VFicus is rather simpler, and so we wished to know whether any costs were the result of replication generally, or from fine-grain control.

5.1 Data overhead

In addition to the data structures already stored by VFicus, selective replication only requires storing the status vector. 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, and therefore there is zero data overhead in these cases.

Most attributes need to be transfered between communicating sites during reconciliation; file data is only transferred if necessary. Specifically, (95 + 20N) bytes are transferred for each file, which includes both UNIX and Ficus attributes. This constitutes a 4N-byte increase above VFicus. Additionally, we employ a *time-based* optimization, reducing the number of files which must be analyzed [6, 17]. If the file is deemed unchanged since the last time the two replicas communicated, only 63 bytes are transferred. Of course, if one of the two communicating sites is not a data-storing replica of the file, no information is transferred at all.

5.2 Experiments

We compared the performance of the selective replication service with both the conventional, unaltered UNIX version on which it is based (SunOS 4.1.1) and VFicus.

We performed three experiments. First, we compared the cost of local data access in UNIX, VFicus, and Ficus with full support for selective replication. Second, we compared the cost of accessing remote data in NFS, VFicus and Ficus. Third, we wanted to have an idea of the cost of selective replication. While the elapsed time to reference a remote replica may well exceed that of a local one due to inherent communication overhead, the hope is that the replication mechanism itself does not add further cost. To measure this cost, we studied data access times in the selective replication service with a variety of different replication patterns.

5.3 Benchmarks

Six benchmarks were used in this evaluation. The first test is the modified Andrew Benchmark (`mab`) [8, 13], which performs a variety of intensive I/O operations followed by a compile phase. The second is a recursive copy test within the replicated namespace, operating on the `/usr/include` hierarchy which on our system contains 4.2 Mb of data and 1191 files, of which 60 are directories. The third through fifth tests are combinations of "find" and "grep": one find, one grep, one find and grep. `Find` and `findgrep` operate on the same `/usr/include` tree; `grep` operates on the `/usr/include/sys` subtree, which is 336Kb of data and 104 files (0 directories). The final test is an "ls" test, which also operates on the same `/usr/include/sys` subtree. To achieve measurable results, many tests execute repeatedly, as listed in Table 2.

Benchmark	Number of Iterations
mab	1
cp	1
find	5
grep	100
findgrep	5
ls	500

Table 2: The number of iterations for each benchmark described in Section 5.3.

All measurements were conducted on Sun IPCs, each with a 207 Mb SCSI disk and standard Ethernet connection. All replicas and sites involved are part of the same Ethernet segment. System load was minimized by isolating machine usage to only the benchmark processes and standard system daemons.

5.4 Results

5.4.1 Local data access cost

We measured the cost of accessing local data for VFicus and Ficus; the results are shown in Figure 5. In the VFicus case, data is stored in a local volume. In the Ficus case, data is stored in a selectively replicated volume with locally stored replicas of all files. The two perform almost identically on all benchmarks. We conclude from this that selective replication does not affect local data access.

UNIX performance is also depicted. While all benchmarks illustrate a performance degradation relative to UNIX, most show only a 6 to 15% performance hit. The `find` benchmark is clearly the worst. This cost is caused by the Ficus implementation: `stat` has historically performed poorly in Ficus due to early design decisions. The method by which Ficus maintains ex-

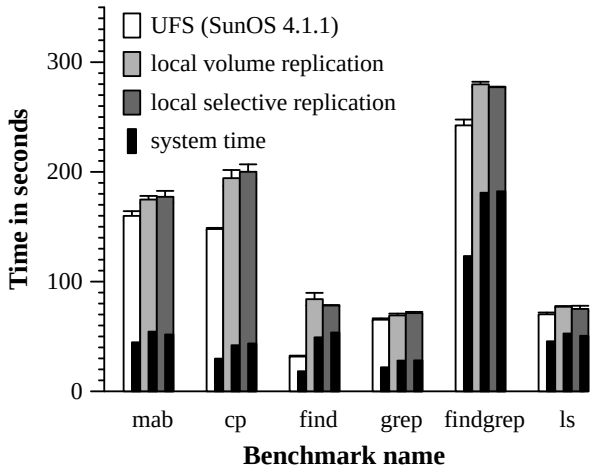

Figure 5: Data access performance for UNIX, VFicus, and Ficus where all data is stored and accessed locally. Elapsed time is indicated by total height, system time by the black bar within. Each data point is the mean of 6 runs. The 95% confidence intervals are indicated, except where insignificant.

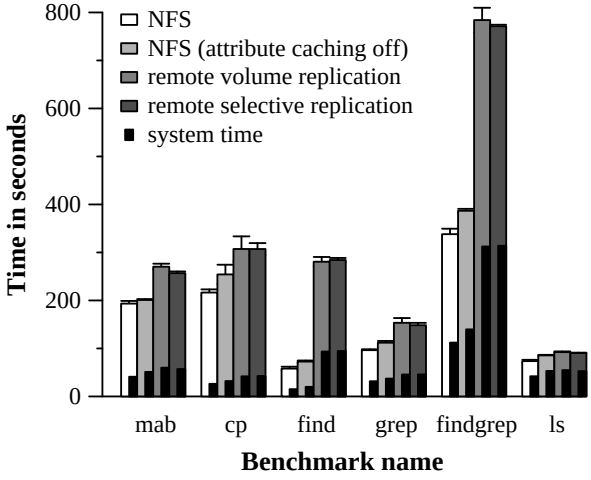

Figure 6: Data access performance for NFS, VFicus, and Ficus where all data is stored and accessed remotely. Elapsed time is indicated by total height, system time by the black bar within. Each data point is the mean of at least 6 runs. The 95% confidence intervals are indicated, except where insignificant.

tended attributes causes at least one and possibly two extra file opens for each open of a user's file. Better extended attribute maintenance can be implemented. Nevertheless, these issues are orthogonal to selective replication and a judgment of its performance.

5.4.2 Remote data access cost

Figure 6 displays the cost of accessing remote data. In the VFicus case, remote data is accessed via a remote volume, mounted into the local tree structure.[7] In Ficus, the top-level directory was locally deleted using the selective replication tools, forcing all data access through the remote replica of the directory. The two NFS cases involve NFS mounting of a remote file system; one used standard NFS, the second removed attribute caching,[8] since VFicus and Ficus do not currently cache attributes.

The results again show that, as expected, selective replication does not cost more than a coarse-grained, volume-based replication service, even when accessing remotely-stored data. Some of the benchmarks (`mab` and `ls`) were even faster with selective replication, due to minor artifacts of the implementation. We actually expect the `ls` benchmark to be faster, because it does not need to access remote data, as explained below.

Compared to NFS, Ficus performed adequately on most benchmarks, with the exception of `find` and `findgrep`. We attribute the poor performance of these benchmarks to the same cause as in the local case above; they are mostly dominated by `stat`-time, and therefore Ficus pays a heavy penalty performing multiple opens for the extended attributes. In general, we saw only a modest overhead compared to NFS, and one which would be radically reduced if we incorporated attribute-caching and better maintenance of extended attributes into Ficus.

5.4.3 Selective replication cost

Finally, we analyzed the cost of using selective replication. Five of the six benchmarks were studied with a variety of replication patterns, ranging from all data stored locally (local replication) to no data stored locally (remote replication). Figure 7 displays the results. The copy benchmark was not relevant because copy operations typically create all new files with identical replication factors.

For the `mab` tests, the replication patterns in the figure represent different portions of the `mab` hierarchy being remotely replicated. The benchmark copies source files into a temporary directory and performs I/O intensive and compilation operations on

[7] A description of the volume mounting procedure is in [4].

[8] Attribute caching was removed by modifying `nfs_getattr_cache`.

them. In replication pattern 1, all of the source files (`mab/itc/fscript`) are remotely replicated. In replication pattern 2, just the binary executables (`mab/bin`) used in the compilation phase are remotely replicated. Replication pattern 3 has just the working subdirectory (`mab/itc/fscript/tmp`) remotely replicated.

For all other benchmarks, the replication patterns 1 to 3 roughly correspond to 10%, 20%, and 40% of the data remotely replicated. Specifically, replication pattern 1 has 7 directories and 406Kb total of the `include` subtree remotely replicated, of which 39Kb is in `include/sys`. Replication pattern 2 has 16 directories and 896Kb total of `include` remotely replicated, of which 76Kb is in `include/sys`. Finally, replication pattern 3 has 21 directories and 1.9Mb total of `include` remotely replicated, of which 112Kb is in `include/sys`. The replication patterns were selected randomly.

As Figure 7 illustrates, the `mab` benchmark shows no performance degradation for selective replication patterns with the compilation and test directory `mab/itc/fscript/tmp` replicated locally. The copying of remote source files and the usage of remote binaries is unnoticeable. When `tmp` is not replicated locally, we experience a modest performance degradation, regardless of where the source files are replicated. Performance is dominated not by copying the files, but by where they reside when the `mab` operations are applied to them.

The `ls` benchmark also shows no performance degradation, though for a different reason. The directory-entry invariant ensures that each directory stores entries for each child in at least an *attributes-only* state. Even though the files themselves are stored remotely, the names are stored locally. Since the `ls` benchmark simply reads the directory, it never requires remote access. The single exception is in the remote replication case, where `include` itself, and therefore `include/sys`, is remotely-replicated. Since remote access is now required, we expect the `ls` benchmark to be slightly slower, as the data indicates.

`Find`, `findgrep`, and `grep` show steady rises in elapsed time as the quantity of remotely stored data increases. For intensive `find` and `grep` operations, execution time is directly proportional to the amount of data remotely replicated. Unlike with the `mab` and `ls` benchmarks, the savings in disk space accompany an increase in execution time. The increase curve illustrates the tradeoffs available to the user.

5.5 Unanswered questions

We would like to quantify more complex aspects of selective replication, such as the intricate interactions between replication factors, connectivity, and message traffic. For instance:

- How many sites need to be queried on average in the multi-ring topology?
- What is the overall savings in network traffic for a given set of volume replicas?

The communication load is highly sensitive to layout specifics, as load depends on both replication factors and locations. Any particular experiment establishes a fixed replication pattern; repeating with different patterns may only provide limited additional insight if the patterns are not expected in practice. We expect high locality of reference to play a significant role in grouping similar sets of files together at similar sets of volume replicas, leading to quite limited and efficient communication patterns in larger configurations. Additional experience, however, is required.

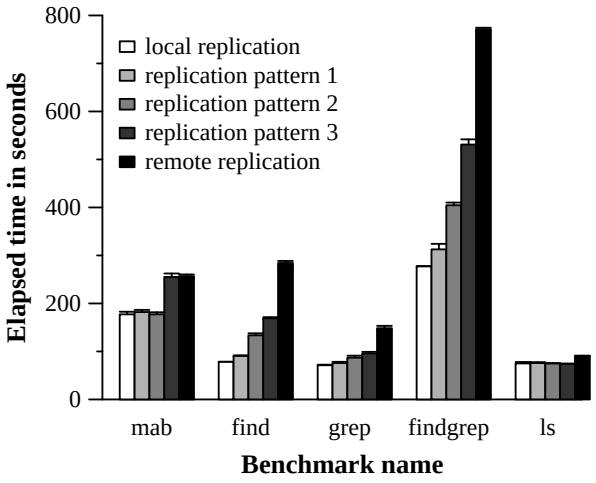

Figure 7: Data access performance for the selective replication service under a variety of different replication patterns. The replication patterns are described in Section 5.4.3. Each data point is the mean of 5 runs. The 95% confidence intervals are indicated, except where insignificant.

6 Related Work

A number of other replication services exist, each with different design choices, assumptions, and levels of service. We will discuss each system briefly.

The Coda file system [9, 20] is an optimistically replicated file system constructed on a client-server model, as opposed to the peer architecture proposed by Ficus.

As such, Coda trivially provides selective-replication control at the clients, but not at the replicated servers, which are, by definition, peers. Additionally, Coda clients cannot inter-communicate due to the inherent restrictions of the client-server model. This restriction dramatically simplifies the algorithms needed to manage consistency, at the cost of limiting the system's utility for mobile workgroups.

Coda has greatly optimized the communications and synchronization between client and server, especially in environments with weak connectivity [10, 12]. Some of the same ideas can and have been applied in Ficus [6]; however, more research is required in optimizing communication protocols for weak connectivity in peer models.

The Little Work project [7] is similar to Coda, but modifies only the clients, leaving the AFS [8] servers unaltered. Congestion caused by client's slow links is reduced in a variety of ways, including client-side modifications of AFS, its underlying RPC, and other congestion avoidance and control methods. However, again clients cannot intercommunicate, hindering the usability of the system. Additionally, the method used to populate the client with the necessary data is suboptimal compared to other strategies [10, 11].

The Bayou system [24] is also a replicated storage system, but based on the peer-to-peer architecture. Like Ficus, Bayou provides support for application-dependent resolution of conflicts. However, unlike Ficus, 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.

Bayou provides *session guarantees* [23] to improve the perceived consistency by users. Additionally, Bayou establishes strong guarantees about its data—writes can be classified either as *committed* or *tentative*. Ficus is working on similar methods of providing stronger consistency guarantees to users [3]. However, Bayou does not support any form of selective replication, so the databases (the Bayou replication unit) must be fully replicated at all storage sites.

The Deceit file system [22, 21], places all files into one "volume" and allows each individual file to be replicated independently with varying numbers of replicas. In this sense it provides selective replication. However, Deceit employs a conservative approach to replication, namely a writer-token mechanism, and as such cannot provide the high availability offered by optimistic mechanisms. In addition, Deceit cannot tolerate long-term network partitions; instead, only a related failure known as a *virtual partition* [22], which eventually corrects itself, is tolerated.

The university LOCUS operating system [16, 25] provides volumes and allows selective replication within the volume. However, the approach taken toward replication is not strictly optimistic. While LOCUS allows concurrent updates across partitions, the mechanism used to merge partitions is complex and does not scale, because it always attempts to maintain a perfect picture of partition membership.

7 Conclusions

Peer models are necessary in a wide variety of environments, but have traditionally provided less flexibility than client-server models. We argue that selective replication provides the replication flexibility of the client-server model while maintaining the advantages of a peer-to-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. The result is an architecture more powerful than either of the two models separately.

Of course, there are more aspects to both models than just functionality and flexibility. The other attributes will need to be addressed before we can definitively conclude that the two models can be merged, maintaining the best aspects of each. Nevertheless, our work with selective replication is a promising step in the right direction.

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] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109–126, Copper Mountain Resort, Colorado, Dec. 1995. ACM.

[2] M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Mar. 1982.

[3] A. Goel. View consistency for optiimistic replication. Master's thesis, University of California, Los Angeles, Feb. 1996. Available as UCLA technical report CSD-960011.

[4] 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. USENIX, June 1990.

[5] R. G. Guy, G. J. Popek, and T. W. Page, Jr. Consistency algorithms for optimistic replication. In *Proceedings of the First International Conference on Network Protocols*. IEEE, Oct. 1993.

[6] 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. IEEE, Nov. 1992.

[7] 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.

[8] 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, Feb. 1988.

[9] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. Technical Report CMU-CS-91-166, Carnegie-Mellon University, July 1991.

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

[11] G. H. Kuenning. Design of the SEER predictive caching system. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994.

[12] 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.

[13] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Conference Proceedings*, pages 247–256. USENIX, June 1990.

[14] T. W. Page, Jr., R. G. Guy, G. J. Popek, J. S. Heidemann, W. Mak, and D. Rothmeier. Management of replicated volume location data in the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 17–29. USENIX, June 1991.

[15] 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.

[16] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A network transparent, high reliability distributed system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 169–177. ACM, Dec. 1981.

[17] 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.

[18] 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. USENIX, June 1994.

[19] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC distributed file system: Principles and design. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 35–50. ACM, Dec. 1985.

[20] 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.

[21] A. Siegel. *Performance in Flexible Distributed File Systems*. Ph.D. dissertation, Cornell, Feb. 1992. Also available as Cornell technical report TR 92-1266.

[22] A. Siegel, K. Birman, and K. Marzullo. Deceit: A flexible distributed file system. In *USENIX Conference Proceedings*, pages 51–61. USENIX, June 1990.

[23] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 140–149, sep 1994.

[24] 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.

[25] 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.

[26] 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.