Log-Based Directory Resolution in the Coda File System

Puneet Kumar
M. Satyanarayanan

December 14, 1991
CMU-CS-91-164

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Optimistic replication is an important technique for achieving high availability in distributed file systems. A key problem in optimistic replication is using semantic knowledge of objects to resolve concurrent updates from multiple partitions. In this paper, we describe how the *Coda File System* resolves partitioned updates to directories. The central result of our work is that *logging* of updates is a simple yet efficient and powerful technique for directory resolution in Unix file systems. Measurements from our implementation show that the time for resolution is typically within 10% of the time for performing the original set of partitioned updates. Analysis based on file traces from our environment indicate that a log size of 2 MB per hour of partition should be ample for typical servers.

This work was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597, the National Science Foundation PYI Award No. CCR 8657907 and NSF Grant No. ECD 8907068, an IBM Corporation Research Initiation Grant, a Digital Equipment Corporation External Research Project Grant, a Bellcore Information Networking Research Grant and the General Electric Company.

Keywords: Optimistic replication, conflicts, network partition, directory resolution, Unix file systems, performance, logging, RVM, file reference traces.

1. Introduction

Optimistic replication is an effective technique for attaining high availability in distributed file systems [3]. The term "optimistic" refers to the fact that concurrent updates are allowed in multiple network partitions. A pessimistic scheme, in contrast, allows updates in atmost one partition. An optimistic strategy provides higher data availability but cannot guarantee data consistency across partitions. Therefore optimistic replication is preferable when closely-spaced sequential write-sharing is rare, and when coping with it is less onerous than being denied update access during network failures. There is substantial evidence to suggest that this combination of circumstances is often present in distributed Unix[1] file systems [7].

A key problem in optimistic replication is detecting when an object has been updated concurrently in multiple partitions, and determining whether those updates can be transparently merged without violating semantic constraints. Concurrent updates that can be merged are called *benign.* Other updates are called *conflicting.* Without semantic knowledge all concurrent partitioned updates to an object must be treated as conflicting, and merged manually by the user. Manual resolution is undesirable because it reduces the overall usability of the system.

An extremely important object, with known semantics, in Unix file systems is a *directory.* We refer to the process of examining replicas of a directory, deducing the set of partitioned updates and merging them using Unix semantics as *directory resolution.* It has two important side-effects. First, benign updates are propagated to all replicas, thus making them identical. Second, directories with conflicting updates are marked unusable and preserved for future manual repair.

In this paper we describe how the *Coda File System* [10, 11] exploits Unix directory semantics to effectively support optimistic replication. The central result of our work is that *logging* of directory updates is a simple yet efficient and powerful technique for directory resolution. An implementation of directory resolution is complete, and is used on a daily basis by a small user community. Measurements from our implementation show that the time for resolution is approximately 10% of the time for performing the original set of partitioned updates. Analysis based on file traces from our environment indicate that a log size of 2 MB per hour of partition should be ample for typical servers.

2. Coda File System

Coda is designed for a typical research and development environment and is intended for applications like electronic mail, bulletin boards, document preparation and program development. It is not intended to be used for applications like databases that exhibit high degrees of fine-grain write-sharing. Coda consists of a large collection of untrusted Unix clients and a much smaller number of trusted Unix file servers. Each client has a local disk and can communicate with the servers over a high bandwidth network. At certain times, a client may be temporarily unable to communicate with some or all of the servers due to a server or network failure.

Clients view Coda as a single, location-transparent shared Unix file system. The Coda namespace is mapped to individual file servers at the granularity of subtrees called *volumes.* At each client, a *cache manager (Venus)* dynamically obtains and caches volume mappings.

[1]Unix is a trademark of AT&T.

Coda uses two distinct, but complementary, mechanisms to achieve high availability. The first mechanism, *server replication*, allows volumes to have read-write replicas at more than one server. This reduces the probability of an object becoming unavailable due to failures. The second mechanism, *disconnected operation*, takes effect when no server is accessible. While disconnected, Venus services file requests by relying solely on the contents of its cache. When disconnection ends, Venus propagates modifications and reverts to server replication.

2.1. Replica Control Algorithm

The set of replication sites for a volume is its *volume storage group (VSG)*. The subset of a VSG that is currently accessible is a client's *accessible VSG (AVSG)*. File system requests are serviced by Venus from its cache. If the cache does not contain the latest copy of an object Venus contacts the AVSG. The protocol for accessing objects from the servers is *read-status-all-data-one / write-all*. Since this protocol has been described in detail earlier [10], we only present a summary here

Read accesses return the latest accessible copy of an object. To service a cache miss, Venus nominates one server from the object's AVSG as the *preferred server* and obtains both data and status information from it. In parallel it obtains status information from other AVSG members. The system call that caused the cache miss returns successfully only if the version information from all AVSG sites is identical. Otherwise the object needs resolution. Validity of the cached objects is maintained by callbacks.

The update protocol, which is executed when a directory is modified or a file is closed after being written, propagates changes in parallel to all accessible replicas. It consists of two phases, COP1 and COP2, where COP stands for *Coda optimistic protocol*. COP1 performs the semantic part of the operation at each AVSG member, such as transferring file contents, making a directory entry, or changing an access list. COP2 distributes a data structure called the *update set*, which summarizes the client's knowledge of who performed the COP1 operation successfully. The update set is used to maintain the version information used during resolution.

2.2. Directory Updates

Coda directories consist of a series of name-identifier pairs that map names to specific objects in the system. Coda supports the Unix interface for creating, removing and changing directory entries as well as modifying individual objects. Directory entries can be inserted via the `creat`, `link` or `mkdir` system calls, removed via the `unlink` or `rmdir` system calls, and changed via the `rename` system call. Unlike Unix, Coda allows hard links only within a directory. Consequently, the Coda naming hierarchy is constrained to be a strict tree rather than an acyclic graph.

Directory updates are *independent* of one another as long as they do not reference the same object. A set of independent updates can be executed in any order resulting in the same final directory state. For example, operations "`mkdir foo`" and "`mkdir bar`" in a directory `baz` commute with one another and result in the directory `baz` containing two new names `foo` and `bar`. By definition, independent directory updates are benign since we are only interested in write-write conflicts.

Directory updates that are not independent are also benign unless they correspond to one of the following situations:

- *Name/Name conflicts*: Two different objects with the same name are inserted in a directory in different partitions.
- *Remove/Update conflicts*: An entry is removed from a directory in one partition but the corresponding object or its descendants are updated in another partition.
- *Update/Update conflicts*: A directory's meta data, such as its access list, is updated in two or more partitions.
- *Rename/Rename* conflicts: An object is moved into different directories in two partitions.

The first three cases were first identified by Guy in the context of the Locus file system [5]. The fourth category does not exist in Guy's classification because his model does not restrict the naming hierarchy to be a tree.

3. Overview of Directory Resolution

Partitioned updates on an object are detected the first time it is accessed after two or more partitions reconnect. If Venus detects a version mismatch amongst the replicas while servicing a cache miss, it alerts the preferred server to perform resolution and pauses. If resolution is successful, Venus retries servicing the cache miss. In this case, resolution is completely transparent to applications and users. The only noticeable effect is a slight delay in the servicing of the system call. If resolution is unsuccessful, Venus returns an error as the result of the system call that generated the cache miss.

Directory resolution is performed entirely on servers, with clients being responsible only for its activation. This dichotomy is crucial to meeting Coda's goal of *scalability* without compromising *security.* Relying on clients to detect partitioned updates eliminates the need for elaborate machinery on servers to keep track of the state of connectivity of other servers. Such machinery has to be present on clients anyway to guarantee coherence. This is consistent with our strategy of enhancing scalability by using client resources rather than server resources wherever possible [12].

A logical extension of this strategy would make clients rather than servers perform resolution. Unfortunately, this would compromise security because the process of resolution may require examination and modification of regions of the file system for which the user at the client performing the resolution has no access privileges. Our assumption that a client is only as trustworthy as its user requires us to perform such operations on servers.

Coda performs resolution *lazily:* although there may be many partitioned updates in a volume, the system only resolves those objects needed to satisfy the triggering system call. An *aggressive* approach to resolution would, in contrast, strive to eliminate all unresolved partitioned updates as soon as partitions reconnect. Our strategy minimizes the latency of systems calls that trigger resolution. It also reduces the peak demands made on servers immediately after recovery from a crash or network partition. Its main drawback is that unresolved partitioned updates may persist until a further crash or partition, thus increasing the chances of stale data being used or a conflicting update being made. A compromise would be to perform resolution lazily when triggered by a client, but to conduct aggressive resolution in the background during periods of low server load. Our usage experience so far with Coda has not indicated the need for such a hybrid policy.

The *resolution subsystem* is responsible for classifying partitioned updates, propagating benign updates, and preserving evidence from conflicting updates. To perform this function, the subsystem maintains data structures at each server and executes a *resolution protocol* involving the AVSG of the object being resolved. We describe the design of the data structures, their use during resolution and the resolution protocol in the following sections.

4. The Resolution Log

Every replica of a volume in Coda is associated with a data structure known as its *resolution log.* Conceptually, a resolution log contains the entire list of directory mutating operations on a replica since its creation. In practice, of course, logs are of finite length and only the tail is preserved. The size of the log is specified when creating a volume, but can be later adjusted by a system administrator.

4.1. Log Storage

Resolution requires log modifications to be made in a fault-tolerant manner. Each modification should be *permanent* as well as *atomic* with respect to the directory update it reflects. We achieve this by placing both the resolution log and directory contents in *recoverable virtual memory* and modifying them within the same transaction. This is implemented using a lightweight transactional package called *RVM* [8].

RVM is a Unix library that supports local, non-nested transactions on data structures mapped into a process' virtual memory. It provides the basic transactional properties of atomicity and permanence by using a NO-UNDO/REDO write-ahead value log that records committed updates to recoverable virtual memory. Periodically, the modifications represented by the log records are applied to the committed image of virtual memory on disk to reclaim space used by those records. By placing the resolution log in RVM, we combine the well-known strengths of operation logging and value logging.

Our decision to associate resolution logs with volumes was motivated by a number of considerations. First, a per-volume log achieves a reasonable balance between resource usage and efficiency. A single log per server would have achieved better utilization of RVM, but would have given us no control over the usage of RVM by individual users. At the other extreme, a per-directory log would have been more efficient since irrelevant entries would not have to be examined during resolution. But that approach would have resulted in much greater internal fragmentation of RVM. A second consideration is that a per-volume log is consistent with Coda's policy of associating disk quotas with volumes. A final consideration is that the operands of system calls in Coda may span directories but not a volume boundary. Consequently, a volume is the smallest encapsulating unit whose log is guaranteed to contain all the information needed to resolve an update.

4.2. Log Format

The organization of the resolution log meets three requirements. First, it makes efficient use of log storage. Second, it supports efficient recording of updates during normal operation, as well as efficient traversal of log entries during resolution. Third, it contains all the information needed to perform resolution.

The first two requirements are met by organizing the log physically on a per-volume basis, but logically on a per-directory basis. The log for a directory is realized as a doubly-linked chain of log entries embedded in the volume log. Recording a directory update consists of finding a free entry in the volume log, linking it to the end of the directory's log, and filling in the fields of the entry. During resolution, it is usually sufficient to examine the log entries of the directory being resolved. Only on rare occasions is it necessary to examine the logs of other directories.

To meet the third requirement, each log entry has to contain the opcode of the corresponding system call, names of new Coda objects created by the call, and the low-level unique identifiers (called *fids*) of all Coda objects created, deleted or modified by the call. In addition each entry contains a *storeid*, which is a client-generated tag that uniquely identifies an update in Coda. Figure 1 shows the log entry for a simple directory operation in Coda such as file creation.

```
typedef struct
 {
 unsigned    serverid;
 ViceStoreId storeid;    /* of this update */
 unsigned    opcode;     /* of this mutation */
 VnodeId     dvnode;     /* fid of this directory */
 long        nextindex;  /* directory log link */
 long        previndex;  /* directory log link */
 }common_log;
```

(a) Prefix of Every Entry

```
struct create_log
 {
 common_log  cl;        /* prefix*/
 char        *name;     /* of new child */
 VnodeId     cvnode;    /* fid of new child */
 };
```

(b) Entry for File Creation

Figure 1: A Simple Log Entry

Log entries for deletions are more complex. They contain the state of the object when deleted to unambiguously detect *remove/update* conflicts during resolution. For a deleted file, the final state is encoded in its *Coda version vector* [10]. For a deleted directory, this information consists of a pointer to its resolution log, as shown in Figure 2a.

The most complex log entry, shown in Figure 2b, corresponds to the `rename` operation. Such an entry is created in each of the logs of the two directories affected by the operation. Since a `rename` may delete an existing target, the log entry contains sufficient information to also detect any ensuing remove/update conflicts.

5. The Resolution Algorithm

Resolution uses the log from each replica to deduce and propagate the set of partitioned updates to all replicas. For this purpose, each replica's log is made available to every member of the AVSG. In Section 5.1, we focus on the actions at a single server. Next, in Section 5.2, we describe how resolution is coordinated among multiple servers. Finally, in Section 5.3, we identify a number of complications that can arise in resolution and show how they can be handled.

5.1. Compensation at One Site

The *compensation algorithm* is executed at each AVSG member after that server has received the log of every other member of the AVSG. For the purpose of this discussion, the server at which the algorithm is executing is called the *local* server; all other AVSG members are called *remote* servers. The goal of the compensation algorithm is to use the logs of all replicas to compute the set of partitioned updates missed by the local server and to apply a sequence of updates to compensate for the missed updates. Detection of

```
struct rmdir_log
  {
  common_log  cl;         /* prefix */
  char        *name;      /* of deleted child */
  VnodeId     cvnode;     /* fid of deleted child */
  int         head;       /* pointer to deleted child's log */
  int         count;      /* length of deleted child's log*/
  ViceStoreId csid;       /* storeid of deleted child */
  };
```

(a) Entry for Directory Deletion

```
struct rename_log
  {
  common_log  cl;         /* prefix */
  unsigned    srctgt;     /* was I source or target's parent?*/
  struct
    {/* info about source */
    char        *oldname;
    VnodeId     cvnode;
  } rename_src;
  VnodeId     OtherDirV; /* fid of other parent */
  struct
    {/* info about target */
    char        *newname;
    int         tgtexisted; /* was an old target deleted? */
    VnodeId     TgtVnode;   /* fid of old target */
    union
      {/* info about old deleted target */
      ViceVersVec   TgtGhostVV;  /* if it was a file */
      struct
        {/* if it was a directory */
        int         head;
        int         count;
        } TgtGhostLog;
      } TgtGhost;
    } rename_tgt;
  };
```

(b) Entry for Rename

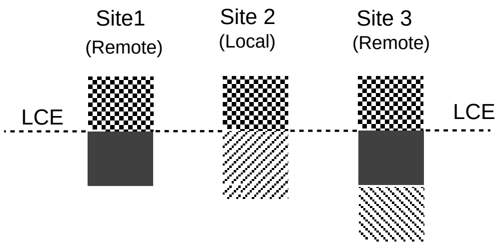
Figure 2: More Complex Log Entries

conflicts, if any, is a side effect of the algorithm. The algorithm proceeds in three steps as shown in Figure 3.

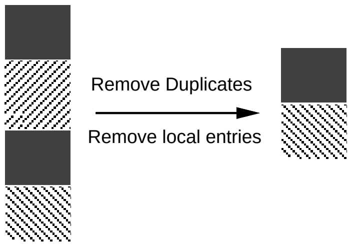
In the first step, the set of all partitioned updates is deduced. This is done by scanning each log backwards starting from the last entry and finding the most recent entry that exists in all logs. This is called the *latest common entry (LCE)*, and represents the most recent point when all the replicas were identical. Resolution relies on the invariant that *entries in each log after the LCE correspond to exactly the set of partitioned updates.* This invariant follows from two observations. First, if entries with the same storeid are found in the logs of a set of replicas, it implies that these replicas successfully participated in the same update. Second, the Coda update protocol guarantees that updates succeed only at replicas that are already identical. Entries prior to the LCE are not used and can be discarded for the subsequent steps of this algorithm.

In the second step, the set of updates missed by the local server are deduced from the set of all partitioned updates. The partitioned updates from each replica's log are merged and the duplicate entries removed. Then the log entries corresponding to partitioned updates already performed at the local server are removed. Due to dependencies between log entries from one server, the merge must maintain their order. For example, the entry for `rmdir foo` must follow the entry for `mkdir foo` because these operations do not commute. But log entries from different servers can be merged in any order.

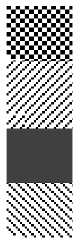
In the third step the updates missed by the local server are executed. These updates modify permanent data structures in RVM and are all performed within a single transaction. If a serious failure such as running out of disk space occurs during the transaction, the entire step is aborted and the algorithm fails. Updates that invert each others' effects are not executed at all. Before executing each update, the server ensures that the resulting state will not violate any semantic invariant. If this is not the case, it marks the



(a) Step 1: Compute LCE



(b) Step 2: Discard useless entries and merge logs



(c) Step 3: Perform operations - Final log state

This figure shows the steps of the compensation algorithm. The algorithm is being executed at *site 2* and the directory is replicated at three servers *site 1*, *site 2* and *site 3*. The shading is different for updates in different partitions. The figures shows (a) the logs made available to *site 2*, (b) how the compensating operations are calculated and (c) the log at *site 2* just after it executes the compensating operations.

Figure 3: Steps of the Compensation Algorithm

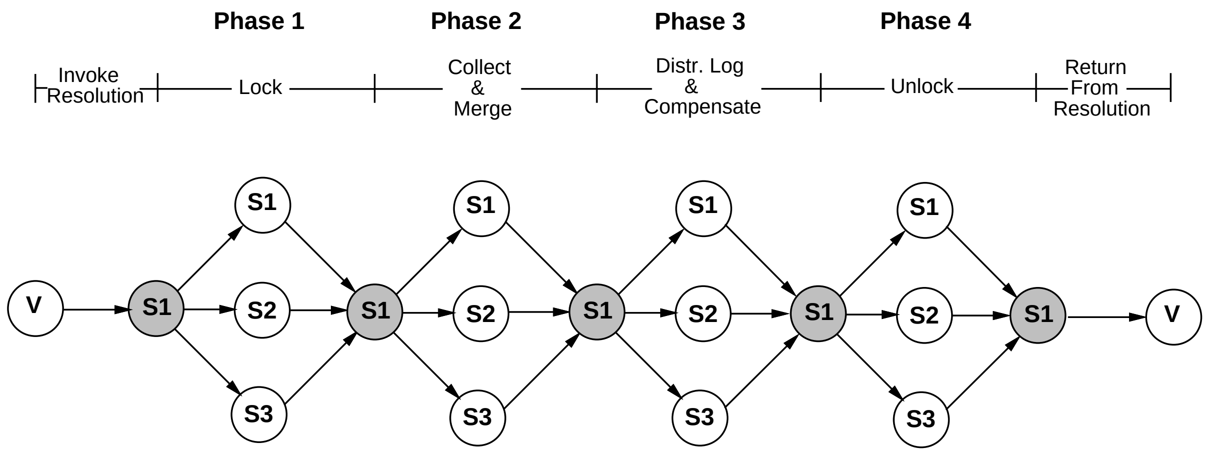
object that was to be modified in conflict. As each update is performed, a log record reflecting this mutation is spooled to the resolution log. Once the entire list of updates has been applied, the encapsulating transaction commits and the compensation algorithm at this site is complete.

5.2. The Resolution Protocol

In this section we describe how resolution is coordinated between multiple servers. The resolution protocol is coordinator-driven, with one AVSG site acting as coordinator and the others acting as subordinates. The resolution protocol proceeds in four phases, as shown in Figure 4. To improve performance, the coordinator uses a parallel RPC mechanism [9] to communicate with subordinates.

The protocol serves two purposes. First, it distributes resolution logs to all AVSG sites so that each can execute the compensation algorithm described earlier. Second, it distributes the final result of resolution to all AVSG sites. Prior to the execution of the protocol, some of the participating replicas may differ from others. At the end of the protocol, either all these replicas are identical and ready for immediate use, or have been marked in conflict and are unavailable until manually repaired.

Our description below describes the events in the absence of failures. If a subordinate fails, the coordinator times out and excludes it from subsequent phases of the protocol. If the coordinator fails, the client times out and restarts the protocol, nominating another coordinator. In all cases, the atomicity of actions at each site is guaranteed by RVM.



This figure shows the sequence of RPCs during resolution. The client *V* invokes resolution by nominating server *S1* as coordinator. The four phases of the protocol are executed at three subordinate servers *S1*, *S2* and *S3*. The node labeled *S1* is shaded when the server is acting as coordinator and unshaded when it is acting as subordinate. If a conflict is detected in phase 3, it is distributed via an extra RPC before phase 4.

Figure 4: Phases of the Resolution Protocol

5.2.1. Phase 1: Locking

Resolution begins with the coordinator requesting each VSG site to lock its replica of the volume containing the directory being resolved. The sites that respond to this request become the subordinates of the resolution protocol; other sites are ignored in the rest of the protocol. All responding sites must indicate successful lock acquisition; otherwise the protocol is aborted and an error code returned to the triggering client.

A resolution lock excludes all other mutations on a replica, including those from normal updates, manual repairs or any other instances of directory resolution in that volume. But non-mutating operations such as reading a file or listing a directory in the volume are permitted. Although locking at finer granularity would improve concurrency, it would be more complex to implement. Our experience so far suggests that this complexity is not warranted.

The resolution lock is held for the entire duration of the protocol, and times out in the event of a coordinator crash or network failure. The value of this timeout has to be greater than the longest expected resolution time, and is set conservatively to 10 minutes in our implementation.

5.2.2. Phase 2: Log Collection and Merging

In this phase, the log entries needed for resolution are collected by the coordinator. Each subordinate first extracts the log of the directory being resolved from its volume log. It then scans the extracted log, composes a list of other objects whose logs might also be needed, and extracts those logs recursively. For example, if a subtree is deleted during a partition, the logs of all the directories in the subtree are needed to resolve its parent. The coordinator merges the logs received from all the subordinates into a linear data structure that preserves the identification of each log.

5.2.3. Phase 3: Log Distribution and Compensation

This phase begins with the coordinator sending the merged logs to subordinates. At this point, each subordinate has logs from every accessible replica, and can perform the compensation algorithm described in Section 5.1. Each subordinate returns a list of conflicts, if any, that arose during this phase.

Although resolution may be successful at a subordinate, the fate of resolution at other sites is still unknown. As a precaution against premature termination of the protocol due to failures, each subordinate marks its replica with a unique storeid. This ensures that any future comparison involving the replica in its current state will trigger resolution again.

Having each subordinate compute its own compensating operations exploits the parallelism inherent in this task. This opportunity would have been lost, had we chosen the alternative strategy of having the coordinator compute the compensating operations for each subordinate. But the latter approach would have involved less data transfer, since the coordinator would have shipped compensation lists rather than the larger merged logs.

5.2.4. Phase 4: Unlocking

In the normal case, phase 3 succeeds at all subordinates. The coordinator sends out a new storeid in phase 4, thus marking all the replicas as equal. The subordinates release their resolution locks, and the coordinator returns to the client.

If the return code to phase 3 from any subordinate indicates conflict, the coordinator executes an additional step in the protocol to distribute conflict information to all subordinates. Phase 4 then merely consists of releasing resolution locks, and returning control to the client with an error indicating a conflict.

5.3. Complications

5.3.1. Coping with Finite Logs

Our discussion so far has ignored the fact that log space is finite. Coda keeps log lengths to a minimum by discarding, at the earliest opportunity, portions of logs that will never be needed in future resolutions. Once an update has been reflected at all replicas, its log entry will become the LCE for any future resolutions. Hence older entries can be discarded, resulting in a log with just a single entry. Confirmation that an update has been propagated to all replicas is available from two sources. In normal operation, the COP2 phase of the update protocol distributes this information. During resolution, the coordinator distributes this information in Phase 4. Logs grow only when some replicas are inaccessible, as reported by either of these sources.

What does a server do when a log becomes full? One approach would be to disallow updates to that volume until resolution is done. The other approach, used in Coda, is to allow updates to continue by overwriting entries at the head of the log. This causes the LCE to be lost, a condition that will be detected by the compensation algorithm of any future resolution and reported as a conflict. The Coda strategy enhances update availability and provides an easily-understood tradeoff between resource usage and usability: the larger a log, the lower the likelihood of having to resort to manual repair. However, it would be a simple matter to make the choice between disallowing updates and overwriting log entries a volume-specific parameter.

5.3.2. Resolving with Partial VSG

When resolution proceeds without all VSG members, partitioned updates must be repropagated when other members become accessible. To prevent a site from performing the same operation twice, Coda logs updates during resolution with the storeid of the original update. The log entry contains the same information as the original update's entry to ensure correctness of future resolutions even if the site where the original update was performed becomes inaccessible.

Log entries spooled during resolution do not provide the same guarantee as that provided by entries for client initiated updates: if two replicas' logs have the same log entry, the replicas need not have been identical at that point. Therefore, step 1 of the compensation algorithm that computes the LCE ignores log records spooled during resolution. To make these two kinds of entries distinguishable, log entries spooled during resolution use special opcodes.

5.3.3. Manual Repairs and Resolution

Manual repairs allow the user to perform arbitrary operations at each replica. Once a replica is repaired, its log is truncated and a log entry reflecting the repair is spooled. The storeid for this entry will be the LCE in future resolutions. If a repair is performed when some VSG members are missing, future resolutions triggered by the recovery of missing VSG members will fail because no LCE will be found. Hence the user will have to manually repair the object again. Only a repair performed when all VSG members are up will restore the ability to perform transparent resolution.

5.3.4. Cross-Directory Renames

A rename operation may involve directories far apart in the naming hierarchy. It is necessary to resolve both the source and target parents simultaneously because each may be dependent on other partitioned renames. To correctly handle these cascaded dependencies, the transitive closure of all directories affected by a sequence of renames must be resolved together.

Analysis of file system traces from our environment shows that less than 3% of all directory updates are cross-directory renames. In the light of their relatively rare occurrence, we have chosen not to address transparent resolution of cross-directory renames in our current implementation. But we do guarantee detection of such renames, and mark both parents in conflict. The next version of our system will support this missing functionality.

6. Evaluation

A log-based approach to directory resolution incurs time and space overheads. The time overhead occurs mainly during resolution, with logging being an almost negligible contributor in our implementation. The space overhead arises from the need to maintain logs at servers. The rest of this section answers the two obvious questions that follow from these observations:

- How well does resolution perform?
- How fast does the log grow during partition?

6.1. Performance of Resolution

6.1.1. Metric

A fair estimate of the overhead due to resolution must account for the fact that resolution will take longer when there are more partitioned updates to resolve. Hence the metric we use in our evaluation is the ratio of two times: *resolution time* and *work time.* Resolution time is the elapsed time between detection of a partitioned update and return of control to the client after successful resolution. Work time is the sum of the elapsed times for performing the original set of partitioned updates.

Resolution time is perceptible to the first user to access a directory after the end of a network failure that resulted in resolvable partitioned updates. The elapsed time for failed resolution is less important, since it is swamped by the time for manual resolution.

An increase in partitioned activity lengthens phases 2 and 3 of the resolution protocol. Phase 2 takes longer because larger logs are shipped to the coordinator. Phase 3 takes longer because of an increase in the transmission time to ship a larger merged log to the subordinates, and because of an increase in the times at the subordinates for computing and applying compensating operations. An increase in the number of replicas also increases resolution time because communication overheads are higher, and the computing of compensating operations by subordinates takes longer.

6.1.2. Experiment Design

To quantify the above effects, we conducted a series of carefully controlled experiments using a synthetic benchmark. One instance of the benchmark, referred to as a *work unit*, consists of 104 directory updates. The execution of a work unit proceeds in three steps:

- creation of 20 new objects, consisting of 14 files, 4 subdirectories, 1 link and 1 symbolic link. These numbers approximate the observed composition of typical user directories in our environment.
- simulation of editor activity on the newly-created files. This is done by creating, then removing, a checkpoint file for each.
- simulation of C++ compiler activity on the newly-created files. For each such file, *foo.c,* a file *foo..c* is created; next, a file *foo..o* is created, then renamed to *foo.o;* finally *foo..c* is removed.

An experiment consists of first measuring the work time for performing a variable number of work units on each of *n* partitioned replicas of a directory. Then the partitions between the replicas are healed, resolution is triggered, and the resolution time is measured.

We performed two sets of experiments, one involving partitioned work only at one replica, and the other involving partitioned work at all replicas. In each set, we examined configurations involving 2, 3 and 4 replicas. For each configuration, we varied the load from 1 to 10 work units.

6.1.3. Results

Table 1 and 2 present the means and standard deviations of work and resolution times observed in three trials of each experiment. They also indicate the contributions of individual phases to total resolution time. The tables indicate that resolution time increases primarily with load, and secondarily with the replication factor.

Rep Factor	Load	Work Time (seconds)	Resolution Time (seconds) Total	Phases 1+4	Phase 2	Phase 3	Res Time / Work Time
2	1	31.39 (1.61)	2.43 (0.09)	0.18 (0.01)	0.24 (0.00)	2.01 (0.09)	7.74%
	2	63.47 (3.07)	5.05 (0.32)	0.29 (0.02)	0.50 (0.15)	4.26 (0.16)	7.96%
	3	96.24 (2.99)	8.02 (0.50)	0.45 (0.02)	0.68 (0.12)	6.88 (0.54)	8.33%
	5	155.55 (2.70)	13.91 (1.00)	0.96 (0.60)	1.63 (0.38)	11.32 (0.13)	8.94%
	7	223.81 (1.40)	24.50 (3.96)	1.19 (0.35)	3.67 (2.22)	19.63 (1.44)	10.95%
	10	320.21 (4.63)	47.37 (4.59)	2.22 (0.22)	5.94 (2.43)	39.20 (2.37)	14.79%
3	1	30.41 (0.22)	2.83 (0.17)	0.24 (0.03)	0.31 (0.03)	2.28 (0.17)	9.31%
	2	59.82 (0.05)	5.23 (0.17)	0.34 (0.02)	0.49 (0.12)	4.39 (0.20)	8.74%
	3	89.84 (0.69)	8.06 (0.28)	0.54 (0.09)	0.68 (0.12)	6.83 (0.34)	8.97%
	5	161.62 (2.60)	15.45 (1.14)	0.77 (0.06)	1.59 (0.49)	13.08 (0.63)	9.56%
	7	227.52 (3.77)	25.20 (1.56)	1.27 (0.49)	3.05 (1.24)	20.88 (0.16)	11.08%
	10	331.98 (7.00)	50.68 (1.57)	2.85 (0.42)	8.58 (1.78)	39.25 (2.11)	15.27%
4	1	31.33 (2.82)	2.93 (0.36)	0.25 (0.06)	0.27 (0.03)	2.40 (0.32)	9.35%
	2	59.61 (1.37)	5.47 (0.14)	0.34 (0.01)	0.43 (0.00)	4.70 (0.14)	9.18%
	3	90.82 (0.83)	8.18 (0.11)	0.54 (0.01)	0.63 (0.01)	7.01 (0.12)	9.01%
	5	171.01 (28.92)	17.92 (2.93)	0.79 (0.11)	3.78 (2.37)	13.35 (0.93)	10.48%
	7	264.35 (38.88)	28.04 (2.17)	1.68 (0.19)	3.57 (1.78)	22.79 (0.80)	10.61%
	10	344.80 (31.19)	51.54 (1.50)	3.78 (0.46)	6.68 (0.99)	41.08 (2.14)	14.95%

This data was obtained using a Decstation 3100 with 16MB of memory as client, and IBM APC-RTs with 12MB of memory as servers communicating over an Ethernet. The numbers presented here are mean values from three trials of each experiment. Figures in parentheses are standard deviations.

Table 1: Resolution Time After Work at One Replica

Rep Factor	Load	Work Time (seconds)	Resolution Time (seconds) Total	Phases 1+4	Phase 2	Phase 3	Res Time / Work Time
2	1	61.64 (1.18)	5.35 (0.06)	0.27 (0.03)	2.37 (0.03)	2.70 (0.08)	8.68%
	2	123.56 (0.48)	8.78 (0.44)	0.59 (0.08)	2.64 (0.07)	5.55 (0.46)	7.11%
	3	183.57 (5.18)	15.55 (1.89)	0.90 (0.21)	3.56 (0.85)	11.08 (1.22)	8.47%
	5	321.16 (10.36)	31.32 (0.44)	1.96 (0.17)	7.61 (2.79)	21.75 (2.24)	9.75%
	7	437.46 (6.04)	58.85 (1.58)	5.04 (0.55)	15.70 (0.35)	38.11 (1.97)	13.45%
	10	594.86 (4.24)	82.00 (6.35)	6.38 (1.33)	17.09 (6.72)	58.52 (4.12)	13.78%
3	1	96.73 (1.87)	8.62 (0.59)	0.42 (0.02)	2.68 (0.10)	5.52 (0.58)	8.91%
	2	188.51 (0.87)	15.20 (0.96)	0.66 (0.01)	3.57 (1.10)	10.97 (0.15)	8.06%
	3	292.87 (3.68)	29.04 (3.42)	1.61 (0.60)	4.92 (1.39)	22.51 (2.92)	9.92%
	5	478.54 (13.61)	67.74 (2.81)	2.53 (0.46)	9.19 (1.33)	56.02 (3.77)	14.16%
	7	672.99 (8.08)	122.28 (10.07)	8.87 (1.64)	18.26 (3.73)	95.12 (5.34)	18.17%
	10	990.83 (24.28)	260.53 (20.06)	18.55 (5.36)	40.38 (8.81)	201.54 (14.89)	26.29%
4	1	122.32 (1.37)	11.14 (0.42)	0.53 (0.02)	2.80 (0.12)	7.81 (0.36)	9.11%
	2	251.61 (1.41)	27.43 (0.57)	1.89 (1.35)	5.81 (1.41)	19.73 (1.15)	10.90%
	3	387.06 (5.13)	54.93 (0.18)	3.92 (1.99)	11.09 (0.95)	39.91 (1.22)	14.19%
	5	661.29 (7.47)	144.86 (7.80)	9.35 (2.37)	19.69 (5.95)	115.80 (6.51)	21.91%

This data was obtained from experiments using the same hardware configuration as for Table 1. The numbers presented here are the mean values from three trials of each experiment. Figures in parentheses are standard deviations.

Table 2: Resolution Time After Work at All Replicas

The primary conclusion to be drawn from this data is that a log-based strategy for directory resolution is quite efficient, roughly taking about 10% of the work time. Although the ratio of resolution to work time does rise at higher loads and replication factors, it remains below 15% in all but three of the cases. The highest overhead of about 26% occurs at a workload of 10 at each of 3 replicas. This corresponds to over 1000 updates being performed on each of 3 replicas of a directory.

The tables show that phases 1 and 4 contribute very little to the overall resolution time. Since these phases merely do locking and unlocking, the time for them should be independent of load. But, as a sanity check in our current implementation, the coordinator collects the replicas to verify equality before unlocking in Phase 4. This accounts for the dependence of this phase on load and replication factor in our experiments.

Phase 2 consists of extraction and shipping of logs by subordinates. The time for this is dependent on the total lengths of the logs, which is only related to the total amount of work. This is most apparent in Table 1 where the time for phase 2 increases noticeably with load but hardly varies with increased replication. This observation can also be made indirectly from Table 2. For example, the phase 2 time of 3.56 seconds for a replication factor of 2 and a load of 3 is virtually identical to the time of 3.57 seconds for a replication factor of 3 and a load of 2. Similarly, the phase 2 time of 17.09 seconds for a replication factor of 2 and a load of 10 comes within the bounds of experimental error to the time of 19.69 seconds for a replication factor of 4 and a load of 5.

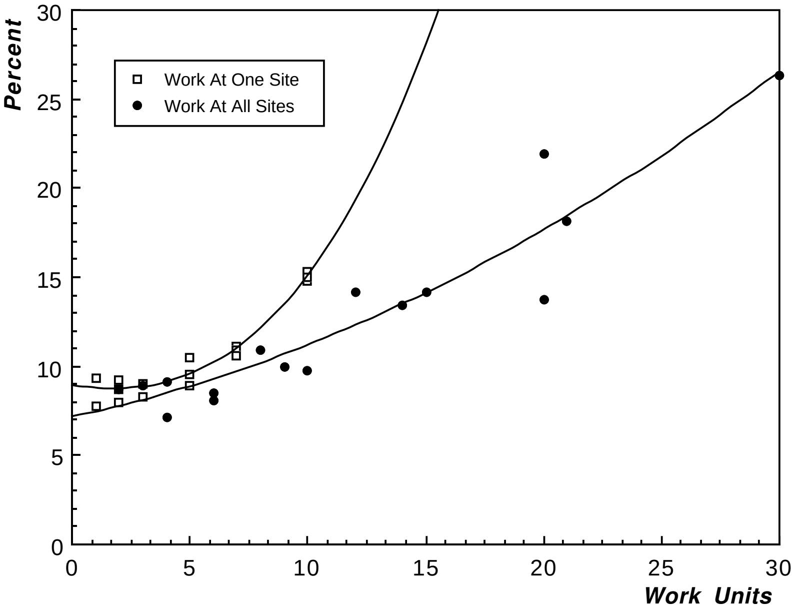
Phase 3 is the dominant contributor to the total time for resolution. This is not surprising, since the bulk of work for resolution occurs here. This includes the shipping of merged logs, computation of compensating operations, and application of these operations. More detailed measurements (not presented here) indicate that the last of these is the most significant component of phase 3 time. This component can be viewed as the cost of replaying missing updates.

The effect of performing resolution in parallel at subordinates can be seen by considering two ways of performing a total amount of partitioned work, WN. When this work is generated at a single replica, WN missing updates must be replayed at each of the other $N-1$ replicas. But when the same work is generated by N replicas each performing work W, only $W(N-1)$ missing updates must be replayed at each replica. Since N is typically small, and since the replay occurs in parallel at all replicas, the time for phase 3 after work at all sites will be lower. If this decrease dominates the increase in time for the rest of the protocol, the total resolution time will also be lower. Figure 5 confirms that this is indeed the case.

6.2. Size of Log

Since a log grows linearly with work done during partition, any realistic estimate of log size has to be derived from empirical data. Our analysis is based on about 4GB of file reference traces obtained over a period of 10 weeks from 20 Coda workstations. The usage profile captured in these traces is typical of research and educational environments.

These traces were used as input to a simulation of the logging component of the resolution subsystem. The simulator assumes that all activity in a trace occurs while partitioned, and maintains a history of log growth at 15-minute intervals for each volume in the system. For each directory update in the trace, the simulator increments the corresponding volume’s log length by the size of the log record that would have been generated by a Coda server. At the end of simulation, the average and peak log growth rates for each volume can be obtained from its history.



For given amounts of total work, this graph compares the ratio of resolution time to work time for two cases. In the first case, work is done only at one site. In the second case, the work is shared equally at all sites. The data points in the upper curve are from Table 1. The X coordinates of a point on this curve correspond directly to its value in the column labeled "Work Load" in the table. The data points in the lower curve are from Table 2. On this curve, the X coordinate of a point is the product of its values in the columns labeled "Rep Factor" and "Work Load" in the table.

Figure 5: Resolving Work Done at One Site versus Work Done at All Sites

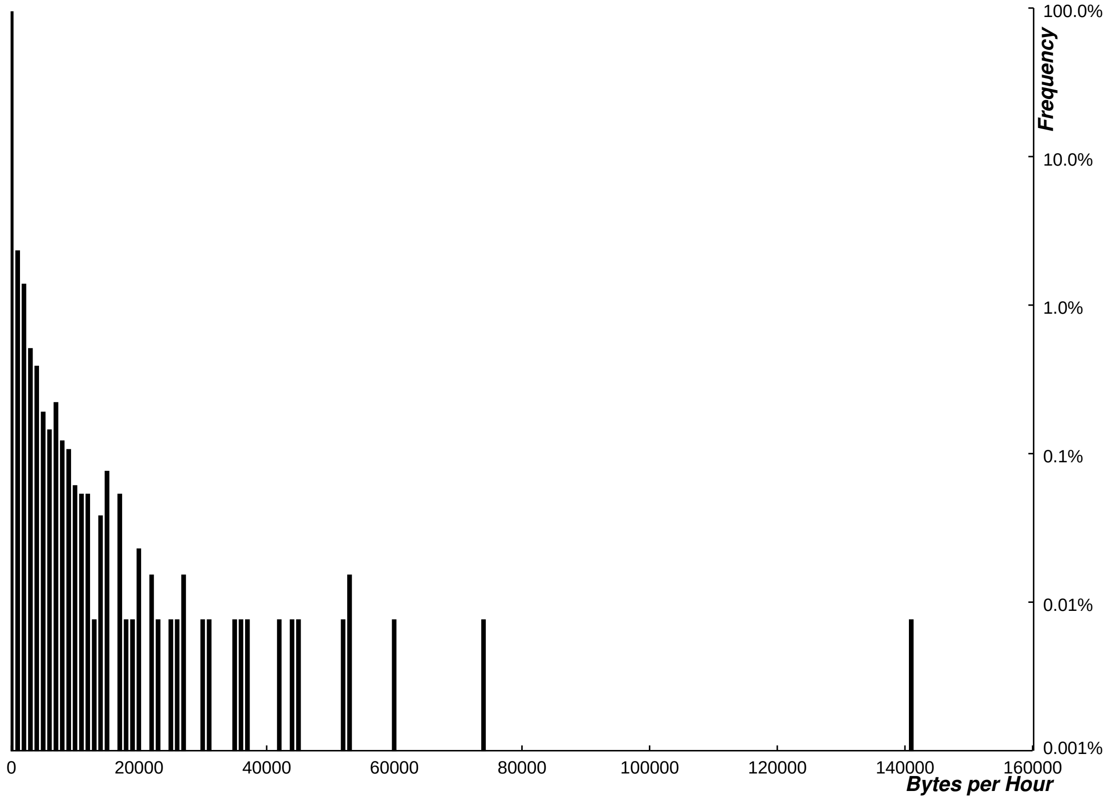
Table 3 shows the distribution of long-term average rate of log growth over all the volumes encountered in our traces. This average is computed by dividing the final log size for a volume by the time between the first and last updates on it. It is clear from Table 3 that long-term log growth is relatively low, averaging about 94 bytes per hour.

Bytes per Hour	Percentage of Volumes
0 to 100	65.91%
100 to 200	20.45%
200 to 300	4.55%
300 to 400	6.82%
400 to 500	2.27%
> 500	0.00%

This data was obtained by trace-based simulation and shows the distribution of long-term average growth rates for 44 volumes over a period of 10 weeks.

Table 3: Observed Distribution of Long-Term Average Log Growth Rates

Focusing only on long-term average log growth rate can be misleading, since user activity is often bursty. A few hours of intense activity during a partition can generate much longer logs than that predicted by Table 3. To estimate the log length induced by peak activity, we examined the statistical distribution of hourly log growth rates for all volumes in our simulation. Figure 6 shows this distribution. Over 94% of all data points are less than 1KB, and over 99.5% are less than 10KB. The highest value observed was 141KB, but this occurred only once.



This histogram shows the distribution of log growth rates for each hour for each volume. Since 44 AFS and Coda volumes were traced over a 10 week period, there are nearly 74,000 data points in this histogram. The width of each histogram bar is1KB. Note that the scale on the vertical axis is logarithmic.

Figure 6: Observed Distribution of Hourly Log Growth

A worst-case design would have to cope with the highest growth rate during the longest partition. A more realistic design would use a log adequate for a large fraction of the anticipated scenarios. Since hourly growth is less than 10KB in 99.5% of our data points, and since an hour-long partition could have straddled two consecutive hours of peak activity, we infer that a 20KB log will be adequate for most hour-long partitions in our environment. More generally, a partition of N hours could have straddled N+1 consecutive hours of peak activity. Hence a log of 10(N+1) KB would be necessary. If a Coda server were to hold 100 volumes (a typical number at AFS installations), the total log space needed on the server would be (N+1) MB.

7. Status and Future Work

Today, Coda runs on IBM RTs, Decstation 3100s and 5000s, and 386-based laptops such as the Toshiba 5200 and IBM PS/2-LX40. A small user community has been using Coda on a daily basis as its primary data repository since April 1990. All development work on Coda is done in Coda itself. As of December 1991 there was nearly 1GB of triply-replicated data in Coda.

A prototype of the resolution subsystem described in this paper has been operational since May 1991. Since RVM was still under development when we began our implementation the resolution logs in the prototype exist only in virtual memory. This also simplifies recovering from bugs in the logging code. Moving the resolution logs into RVM will not invalidate our performance analysis because only the amount of log data spooled during transaction commit will increase, not the number of commits. As a result, we expect the work time as well as the resolution time to increase slightly, but the load dependent behavior to be unchanged.

Our immediate plans are to move the resolution logs from virtual memory to RVM. We also plan to provide support for transparent resolution of cross-directory renames, as discussed in Section 5.3.4. In the longer term, we plan to explore the use of rule-based heuristics for directory resolution. Such heuristics can be exploited by sophisticated applications and end users to customize the resolution of conflicting partitioned directory updates.

8. Related Work

The use of optimistic replication for high availability was explored by a number of researchers in the early 1980s, including Garcia-Molina [4], Blaustein [1], and Davidson [2]. Their work is summarized in the excellent survey by Davidson et al [3]. Most of this work was done in the context of a distributed transactional model, and does not directly apply to Unix file systems.

Locus [13] was the first distributed file system to use optimistic replication and to recognize that Unix semantics could be used for directory resolution. But the proposed ideas were not successfully implemented in the original system. More recently, Guy [6] has developed an implementation of directory resolution in the context of Ficus, a descendant of Locus.

The Coda approach of logging directory updates is conceptually simpler than the Ficus approach of *inferring* these updates from the final states of replicas. The two approaches also differ in their implications for resolution performance. In Coda, performance depends only on the amount of partitioned activity. In Ficus, it also depends on the original size of the directory.

Like Coda, Ficus preserves information about deleted objects in order to detect remove/update conflicts. But the systems differ markedly in their approach to reclaiming space pertaining to these objects. Ficus uses a complex distributed garbage collection algorithm whose scalability is open to question. Coda, in contrast, uses the much simpler strategy of allowing each site to unilaterally reclaim resources via log wrap-around. This provides a clearly-defined trade-off between usability and resource usage, one we believe is essential in any practical system. Finally, we believe that the presence of an explicit log will make it easier to separate policy and mechanism in resolution, thereby simplifying the implementation of heuristic-based resolution.

9. Conclusion

Although conceptually simple, log-based directory resolution has turned out to be more complex to implement than we originally expected. One source of complexity is the need to consider many pathological situations during the computing of compensating operations. Another source is the need to ensure that all steps of the resolution protocol are robust in the face of failures. We have achieved this by making the protocol idempotent. An alternative strategy would have been to use distributed transactions. However, that approach would have required us to run the risk of blocking in case of coordinator failure. It would have also been counter to Coda's general philosophy of using optimistic strategies whenever possible, to improve transparency from the user's perspective.

Our experience with log-based resolution has been highly positive. Our initial concerns about excessive space usage for logging have proved baseless. The speed of resolution is excellent, and is rarely noticeable in normal operation. Overall, we believe that a log-based strategy is indeed appropriate for directory resolution in a distributed file system that supports optimistic replication.

Acknowledgements

The possibility of using logging for directory resolution was first suggested by James Kistler. We are indebted to Lily Mummert for the file reference traces used in Section 6.2, and to Maria Okasaki for the AFS file size statistics used in our directory resolution benchmark. We also wish to express our appreciation to the other members of the Coda project: Hank Mashburn, Gowthami Rajendran, and David Steere.

References

[1] Blaustein, B., Garcia, H., Ries, D.R., Chilenskas, R.M., Kaufman, C.W.
Maintaining Replicated Databases Even in the Presence of Network Partitions.
In *Proceedings of the IEEE 16th Electrical and Aerospace Systems Conference*. September, 1983.

[2] Davidson, S.B.
An Optimistic Protocol for Partitioned Distributed Database Systems.
PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, 1982.

[3] Davidson, S.B., Garcia-Molina, H., Skeen, D.
Consistency in Partitioned Networks.
ACM Computing Surveys 17(3), September, 1985.

[4] Garcia, H., Allen, T., Blaustein, B., Chilenskas, R.M., Ries, D.R.
Data-Patch: Integrating Inconsistent Copies of a Database After a Partition.
In *Proceedings of the 3rd IEEE Symposium on Reliability in Distributed Software and Database Systems*. October, 1983.

[5] Guy, R.G.II.
A Replicated Filesystem Design for a Distributed Unix System.
Master's thesis, Department of Computer Science, University of California, Los Angeles, 1987.

[6] Guy, R. G., Popek, G. J.
Reconciling partially replicated name spaces.
Technical Report CSD-900010, University of California, Los Angeles, April, 1990.

[7] Kistler, J.J., Satyanarayanan, M.
Disconnected Operation in the Coda File System.
ACM Transactions on Computer Systems 10(1), February, 1992.

[8] Mashburn, H., Satyanarayanan, M.
RVM: Recoverable Virtual Memory User Manual
School of Computer Science, Carnegie Mellon University, 1991.

[9] Satyanarayanan, M., Siegel, E.H.
Parallel Communication in a Large Distributed Environment.
IEEE Transactions on Computers 39(3), March, 1990.

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

[11] Satyanarayanan, M.
Scalable, Secure, and Highly Available Distributed File Access.
IEEE Computer 23(5), May, 1990.

[12] Satyanarayanan, M.
The Influence of Scale on Distributed File System Design.
IEEE Transactions on Software Engineering 18(1), January, 1992.

[13] Walker, B., Popek, G., English, R., Kline, C., Thiel, G.
The LOCUS Distributed Operating System.
In *Proceedings of the 9th ACM Symposium on Operating System Principles*. October, 1983.

Table of Contents

List of Figures

List of Tables