# Increasing File System Availability through Second-Class Replication

*James Jay Kistler*[*]
*School of Computer Science*
*Carnegie Mellon University*

## 1. Introduction

In this position paper we describe an important component of our overall approach to achieving high availability in the Coda file system. Coda, a descendant of the Andrew file system (AFS) [6], provides users with shared file access in a large-scale Unix[1] workstation environment. Our goal is to provide users with the benefits of a shared data repository without the negative consequences of total dependence on that repository.

High availability in Coda is achieved through two types of data replication: *first-class* replication, in which entire servers are replicated, and *second-class* replication, otherwise known as client caching. The two are complementary: server replication increases the availability of the entire file store, whereas caching permits operation in the event of total *disconnection*. A particularly valuable use of disconnected operation is the graceful integration of portable computers into the system.

Coda manages replicas, both first- and second-class, *optimistically* [2]. That is, it allows updates to be made at any accessible replica, even in the presence of network partitions. Clients always read from and write to their cache copy of an object, i.e., from/to a second-class replica. Updates are written through to all accessible first-class replicas at file close and at completion of mutating directory operations. First-class sites invalidate second-class replicas via callback messages sent upon their own receipt of an update.

In the absence of failures all replicas, both first- and second-class, will be identical. In the presence of failures replicas may diverge. The system ensures that a client's cache copy is the latest among all accessible first-class replicas, and that all conflicting updates will be detected eventually and the corresponding replicas made available for resolution.

This two-tier replication strategy provides a high-degree of fault-tolerance: a client can continue computing in the face of most failures except that of the client itself. We believe that the consequent weakening of the consistency model seen by users and applications is acceptable for three reasons: the low frequency of write-sharing observed in Unix environments makes conflicts rare in practice, Unix consistency semantics are already weak, and the degree of fault-tolerance provided is substantially higher than any alternative.

In the rest of this paper we focus on our use of second-class replication and its contribution to availability, disconnected operation. Details on server replication, the consistency model, and other aspects of Coda have been described elsewhere [7].

## 2. Disconnected Operation

Disconnected state at a client arises either *voluntarily*, due to detachment of a portable computer, or *involuntarily*, due to failure of all first-class replication sites or intervening

[*]Author supported in part by a Graduate Fellowship from the IBM corporation.

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

communications links. It is not hard to envision a style of disconnected operation where users manually copy files to a local file system on the client workstation prior to disconnection, and manually copy updated objects back to the shared file store upon reconnection. However, this is a decidedly non-transparent arrangement which hearkens back to the early days of distributed file systems. The challenge is to support disconnected operation *transparently*, so that users needn't be aware, or at least concerned, whether they are disconnected or not.

Transparent disconnected operation dictates a much-expanded role for client cache managers, effectively turning them into *pseudo-servers*. We break pseudo-server operation into three phases:

- a *hoarding* phase prior to disconnection. We use the term hoarding to mean cache management geared towards capturing both immediate and future working-sets, rather than just the immediate working-set as in conventional caching. We do not consider hoarding otherwise distinct from caching, nor do we believe that separate caches and hoards should be maintained.[2]

- a *server emulation* phase during disconnection. The pseudo-server must faithfully emulate the semantic and protection checks made by real servers, and it must maintain a detailed enough record of its actions to make the next phase possible and efficient.

- a *reintegration* phase upon reconnection. The pseudo-server must re-sync its state with that of the real servers; this entails conflict detection, automated conflict resolution when possible, and provision for manual resolution when not.

[2]Hoarding is similar to "stashing," as described by Birrell, Schroeder, and Alonso et al [5, 1]. We use our own term to emphasize that hoarding is integral with other caching functions in our system.

Figure 1 illustrates pseudo-server behavior at a high level. In our system the namespace is

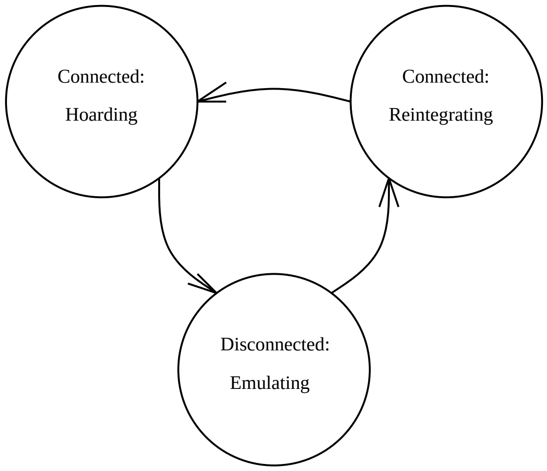

Figure 1: Pseudo-Server State Diagram

subdivided and distributed across many real servers, so a pseudo-server may be in one state with respect to one part and in another state with respect to another. Next, we discuss each of the pseudo-server phases in more detail.

## 2.1. Hoarding

The purpose of hoarding is to enable operation at the client during some future disconnected period(s). Cache misses while disconnected result in suspended or aborted computation; thus, hoarding should result in as many "useful" objects as possible being in the cache when disconnection occurs. The task of hoarding, capturing future working-sets, is complicated by a number of factors:

- working-sets are functions of time intervals; both their starting and ending points are variables.

- future working-sets cannot be predicted perfectly. Predictive ability decreases as both the horizon (i.e., time until start) and the duration of the interval increase.

- involuntary disconnections, both their horizon and duration, cannot be predicted. Voluntary disconnection parameters may be known only approximately.

- client cache space is limited (on the order of tens of megabytes); it is a small fraction of the size of the whole file store.
- the caching needs of connected and disconnected mode are generally at odds with each other. Connected mode utility is maximized by retaining the immediate working-set. Disconnected mode utility, on the other hand, is a complex function of the capture rate of the (infinite) set of future working-sets.

In theory, our hoarding/cache-management algorithm should maximize the user's utility given data on the immediate working-set, future working-sets, future voluntary disconnections, the distribution of future involuntary disconnections, client cache size, and the user's conditional utility function. In practice, of course, we must settle for an approximation that balances the computational expense, the effort involved in collecting the input (e.g., future working-set estimates), and the goodness of the approximation.

Our current algorithm is based on object priorities, where the highest priority candidates are added to the cache until the space allotted to it is consumed. Priorities have two components: a component $s$ representing how recently the object was used at this client, and a component $m$, also called the *hoard priority*, representing the expected future value of the object to the user (0 if unspecified). Users can influence caching behavior in three ways: by assigning hoard priorities to objects, by setting the *horizon* parameter which controls the relative weighting of $s$ and $m$ in the priority calculation, and by setting the *decay* parameter which controls how fast $s$ decreases in response to time since last use.[3]

[3]We assume a *primary-user* model, where the primary user is at the workstation console. Other users may run processes on the machine at the discretion of the primary user, but they cannot substantially affect the hoarding behavior.

## 2.2. Server Emulation

Server emulation involves the temporary promotion of cache copies from second- to first-class status. The goals here are twofold: one is to provide the illusion of connectivity to the user, the other is to enable a smooth (eventual) transition back to connected mode.

The first goal demands that the pseudo-server emulate precisely the actions of real servers in handling file system calls. The illusion breaks down and emulation cannot be performed when cache misses occur—transparency is thus heavily dependent on the success of the previous hoarding phase. A subtle point is that system call handling, and therefore emulation, often involves pathname expansion to derive the actual parameters of a call. If even a single path component is missing from the cache the entire operation is impeded, regardless of whether the target objects themselves are available. To account for such naming effects we manage the cache hierarchically: the object priority function effectively assigns to an interior node the maximum of its own priority and that of all its cached descendents.

Preparation for transition back to connected mode entails logging by the pseudo-server of all mutating operations performed while disconnected. Read operations need not be logged since we only detect and address *write/write* conflicts that arise as a result of disconnected operation.[4] The logging procedure employs several compression and segmentation techniques for efficiency; we omit further discussion of these for the sake of brevity.

One issue that does warrant mention is our use

[4]The Unix file system interface contains no explicit support for transactions; thus, any notion of *read/write* conflict would have to be based on implicit or inferred transactional boundaries. Although we make such inferences with respect to file opens and closes, further inferences seem inappropriate.

of Mashburn's *recoverable virtual memory* (RVM) package[4]. RVM is a lightweight, transaction-like facility which breaks out and addresses separately the basic properties of transactions: atomicity, permanence, serializability. A local, non-nested RVM transaction guarantees only atomicity. Control over permanence is given to the application by allowing it to specify when commit records are forced to the log device. Asynchronous log forces provide dramatic performance benefits, but they sacrifice permanence. Conventional transaction semantics can be achieved by forcing immediately upon every transaction commit. Synchronization, and hence serializability is solely the responsibility of the application.

The pseudo-server stores all persistent data (logs, directories, other cached meta-data) except file contents in RVM. This has two key advantages. One, the atomicity guarantees of RVM greatly simplify the task of managing and recovering pseudo-server state. Two, we can tailor the permanence/performance trade-off to the mode of operation that we are in. When in connected mode we use asynchronous log forces with a relatively low frequency (order of minutes or tens of minutes). This yields very good performance with permanence comparable to a local Unix file system. When we become disconnected we increase the log-force frequency, all the way to synchronous forces if desired, at a cost of increased latency for users.[5]

## 2.3. Reintegration

Reintegration involves the demotion of temporarily promoted cache copies from first-back to second-class status. Disconnected updates must be merged into the shared repository so that they become visible throughout the system.

[5]Our clients are equipped with non-volatile but not necessarily stable storage. Thus, our permanence guarantees all presume the absence of local media failure.

Our merge technique is based on replay of the pseudo-server's log at a first-class replication site. Log replay can be viewed as the write-back part of a write-back caching scheme. However, unlike conventional write-back caching, we have the possibility of update conflicts since we enforce no concurrency control between partitioned replicas. We also have the possibility of protection conflicts since a user's credentials may have expired, and not been renewed, by the time reintegration occurs.[6] The replay algorithm detects conflicts, and attempts to automatically resolve them in two ways: via Unix directory semantics, and via user-supplied heuristics.

Automated resolution using directory semantics is feasible because most operations on Unix directories commute. For example, the partitioned creation of files `foo` and `bar` in the same directory are independent events; it matters not which order they are eventually applied. However, not all directory operations commute, e.g., partitioned creation of files with the same name in a directory, so some cases must be handled differently.

Resolution by heuristic is attempted if automated directory resolution fails or is not applicable. The motivation for heuristic resolution comes from the fact that manual conflict repair is both irksome and (often) mechanistic. Simple rules which break conflicts based on comparison of file names, update times, last authors, etc can be pre-assigned to objects or collections of objects by users. Successful application of a heuristic results in either (1) acceptance of an update from pseudo-server's log and rollback of an operation committed at the first-class replica, or (2) rejection of an update from the pseudo-server's log. The danger of "incorrect" heuristics is mitigated by preserving "discarded" versions in an audit area.[7]

[6]Coda uses authentication tokens, similar in spirit to Kerberos tickets[8], which have fixed lifetimes (on the order of a day).

[7]Heuristic resolution is an instance of the "data-

When neither automated directory nor heuristic resolution succeeds in settling a conflict, the system prepares for eventual manual repair. The inconsistency is contained by making the uppermost node involved in the conflict inaccessible to normal file system operations, and by moving the divergent branches to a reserved place at the server. A user can manually repair the object(s) using a special tool provided for this purpose. The repair tool is able to locate the various conflicting branches—there may be arbitrarily many—and superimpose a sensible naming structure over them. Commitment of the repair makes the affected node(s) generally accessible again.

## 3. Status and Future Work

We are in the midst of implementing and evaluating disconnected mode for Coda. Basic support for hoarding, emulation, and reintegration has been completed, although some of what has been described in this paper remains to be done. The system is in daily use by the members of our research group, and we are acquiring portable workstations so that we can get experience with voluntary disconnected operation.

As yet unimplemented functionality includes working-set monitoring and specification tools, integration with RVM, and heuristic resolution. Our future work consists of adding this functionality and measuring and analyzing the performance and availability characteristics of the system.

## References

[1] Alonso, R., Barbara, D., and Cova, L. Using Stashing to Increase Node Autonomy in Distributed File Systems. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems* (October 1990).

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

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

[4] Mashburn, H. *RVM: Recoverable Virtual Memory*, 0.1 ed. Carnegie Mellon University, June 1990.

[5] Needham, R., and Herbert, A. Report on the Third European SIGOPS Workshop, "Autonomy or Interdependence in Distributed Systems". *Operating Systems Review 23*, 2 (April 1989).

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

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

[8] Steiner, J., Neuman, C., and Schiller, J. Kerberos: An Authentication Service for Open Network Systems. In *Proceedings of the Winter Usenix Conference, Dallas* (February 1988).

patch" philosophy promoted by Garcia-Molina et al[3]. The FACE group at Princeton is also applying this technique to partitioned file merge[1].