Transparent Disconnected Operation for Fault-Tolerance

Position Paper

James Jay Kistler
School of Computer Science
Carnegie Mellon University
Email: jjk@cs.cmu.edu

February 24, 1992

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) [4], provides users with shared file access in a large-scale Unix[1] workstation environment. Our philosophy is to provide users with the benefits of a shared data repository without the negative consequences of total dependence on that repository.

The primary technique for achieving high availability is, of course, replication. Coda uses two types of replication: *first-class* replication, in which entire servers (or logical subsets of them) are replicated, and *second-class* replication, otherwise known as client caching. The two types are complementary: server replication increases the availability of the entire file store, whereas caching permits operation in the event of *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 cached copy of an object. 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 (a) that a client's cache copy is the latest among all accessible first-class replicas, and (b) 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: (1) the low frequency of write-sharing observed in Unix environments makes conflicts rare in practice, (2) Unix consistency semantics are already weak, and (3) the degree of fault-tolerance provided is substantially higher than any alternative.

In the rest of this paper we focus on our design for and initial experience with disconnected operation. Details on server replication, the consistency model, and other aspects of Coda have been described elsewhere [5].

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 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 then 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.

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

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 hoarding to mean cache management geared towards capturing both short- and medium-term working-sets, rather than just short-term as in traditional caching. We do not consider it 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 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, entailing the propagation of disconnected updates and the invalidation of stale cache copies. Conflict detection must be ensured, and provision for conflict resolution made.

Figure 1 illustrates pseudo-server behavior at a gross level. Note that the namespace will likely be subdivided and

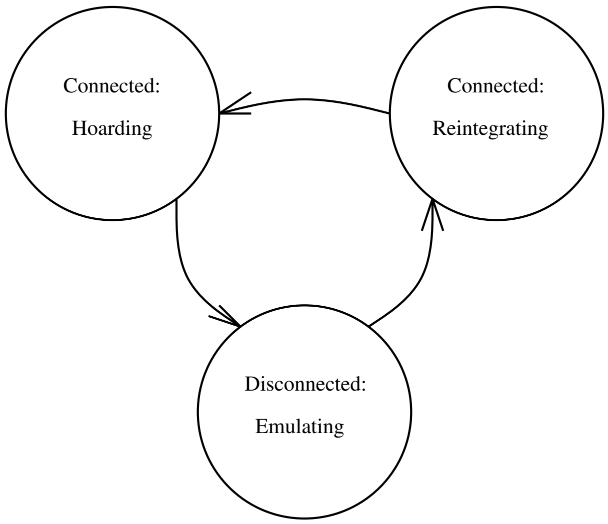

Figure 1: Pseudo-Server State Diagram

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. Also, the two connected states are not necessarily exclusive; hoarding may begin before reintegration completes, subject to internal synchronization constraints. Next, we discuss each of the pseudo-server phases in slightly more detail.

The purpose of hoarding is to maximize the cache hit ratio during future disconnections. If disconnections are short, the usual caching policy of on-demand fetch and LRU replacement may be adequate. However, since we wish to support long disconnections, say up to a week in duration, estimates of the medium-term working-set clearly are needed. Such information can only come from the user. To make this as painless as possible, we allow the user to specify a set of "interesting" files, with the degree of interest represented by a *hoard priority*. The cache manager must balance the caching needs of connected and disconnected operation, that is, try to satisfy both short- and medium-term goals. It uses a strategy reminiscent of certain prioritized scheduling algorithms. The current priority of a cached object is a function of its hoard priority, possibly zero, and a metric representing how recently (or often) the object was used. Files are brought into the cache on demand (i.e., on cache miss), and also as a result of periodic pre-fetch when the priority of interesting, uncached files exceeds that of the lowest-priority cached objects. Coda uses whole-file caching

[2] Hoarding is similar to "stashing," as described by Birrell, Schroeder, and others [3, 1]. We use our own term to emphasize that hoarding is integral with other caching functions in our system.

both because it has been shown to increase scalability and because it provides additional resiliency for disconnected operation (i.e., once a file is opened, reads and writes can always be satisfied).

Server emulation is the temporary promotion of cached copies from second- to first-class status. This transition has several important consequences. First, the promotion is only temporary. Cache copies are still second-class in the eyes of the system; therefore, all update operations will have to be revalidated with respect to integrity and protection before acceptance at first-class sites. Thus, it behooves the pseudo-server to be as faithful as possible in its emulation. Second, the pseudo-server has no recourse to first-class sites while disconnected, so it had better have hoarded <u>everything</u> it will need while it was connected. In particular, this means that the complete naming (i.e., ancestor directories) and associated protection information for a file must be in the cache for the file to be usable. This induces a hierarchical cache structure, which is considerably more difficult to manage than a flat structure where each object can be treated independently. The absence of first-class sites during disconnected operation also means that pseudo-servers must be capable of generating low-level names to support object creation. Third, the pseudo-server must prepare for graceful, transparent return to second-class status. It must log its actions while disconnected so that conflicts can be detected at reintegration, and so that non-conflicting operations can be quickly replayed. Fourth, the pseudo-server must be able to cope with resource exhaustion, i.e., the cache filling up with "dirty" files and their access paths. The easiest way to handle this is to prohibit further updates until reintegration has freed up some space. More difficult, but probably more appealing to users, is to allow selective "back-out" of previously executed updates.

Reintegration is the demotion of temporarily promoted cached copies from first- back to second-class status. The transition must involve the preservation of user modifications somewhere. In the normal case, this will be back in the shared repository, and the process will be entirely transparent. There are two cases in which transparent reintegration is not possible. One, revalidation fails because the disconnected user's authentication tokens have expired[3]. Two, the system detects conflicts (i.e., serialization errors) which it cannot automatically resolve in the course of playback[4]. Such conflicts must be resolved manually by a user. Failed reintegration causes (1) the uppermost node involved in the conflict to be made inaccessible to normal file system operations, and (2) the state at the pseudo-server (i.e., the modified sub-tree) to be *migrated* to a reserved place at the server. The user can then manually repair the conflict using a special tool which is provided for this purpose. The repair tool is able to locate the various conflicting branches of the object at the server (there may be arbitrarily many), and superimpose a sensible naming structure over them. Its last act is to apply the fix chosen by the user and make the repaired node generally accessible again.

3. Conclusion

We consider the ability to operate while disconnected a significant advantage in a distributed file system. In the presence of large-scale, where normal file service is dependent upon the correct functioning of many anonymous components, one is dumped into disconnected state with irritating frequency. Server replication provides a degree of fault-tolerance, but it is expensive and there are always cases where it does not suffice. An obvious and important instance of this is a portable computer which is periodically detached from the network. Of course, manual copying to and from a client's local file system provides the basic service, but it is a primitive and unappealing solution. Since clients already cache objects in many distributed file systems for performance reasons, it makes sense to overload this function for fault-tolerance. We can use processing power at the client, which is the most readily available resource in the system, to provide a robust and generally very transparent service.

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

[4] Automatically resolvable conflicts include most directory modifications, for example, partitioned creation of uniquely named files in a directory.

References

[1] Alonso, R., Barbara, D., and Cova, L. Augmenting Availability in Distributed File Systems. Tech. Rep. CS-TR-234-89, Princeton University Department of Computer Science, October 1989.

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

[3] 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).

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

[5] 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).