Partially Connected Operation

L.B. Huston
http://www.citi.umich.edu/u/lhuston

P. Honeyman
http://www.citi.umich.edu/u/honey

Center for Information Technology Integration
University of Michigan
Ann Arbor

ABSTRACT

RPC latencies and other network-related delays can frustrate mobile users of a distributed file system. Disconnected operation helps, but fails to use networking opportunities to their full advantage. In this paper we describe *partially connected operation*, an extension of disconnected operation that resolves cache misses and preserves client cache consistency, but does not incur the write latencies of a fully connected client. Benchmarks of partially connected mode over a slow network indicate overall system performance comparable to fully connected operation over Ethernet.

1. Introduction

An important advantage of a distributed computing environment is on-demand access to distributed data. Disconnected operation [11, 7], a form of optimistic replication that allows access to cached data when file servers are unavailable, has proved successful at providing this access to mobile users. Disconnected operation is especially successful at hiding network deficiencies by deferring and logging all mutating operations, replaying them later.

Distributed systems tend to be designed to work in environments that provide high data rates and low latencies, but these assumptions are generally invalid in a mobile environment. Here, disconnected operation has broad applicability, but is something of a blunt instrument: because it treats networks as either available or unavailable, disconnected operation does not account for the varying degrees of network quality encountered by mobile users.

For example, even though AFS [6] caches aggressively and has good support for low-speed networking in the transport protocol [1], the network latency that accompanies many operations can make AFS over a low-speed network a trying experience. This affects user satisfaction when interactive response time is increased beyond that which a user is willing to tolerate.

One option is to use disconnected operation when only a low bandwidth network is available, using the network solely to satisfy cache misses. This approach does not support the AFS cache coherence mechanisms, so a user may unwittingly use stale data at a time when it is possible to obtain the most recent version. Furthermore, mutating operations are not propagated immediately, increasing the chance that two users might concurrently update the same file.

Lying between connected and disconnected operation is a mode of operation that allows us to hide many of the network latencies, but to continue to use the network to maintain a relaxed form of cache consistency. In the remainder of this paper, we give an overview of our approach and some implementation details, and present some benchmarks that illustrate the effectiveness of the technique.

2. Background

The work presented in this paper is based on a version of the AFS client that supports disconnected operation [7]. The client cache manager supports three modes of operation; connected, disconnected, and fetch-only. In connected mode the cache manager is an ordinary AFS client, using callback promises to preserve cache coherence [10]. In disconnected mode the cache manger treats the network as unavailable, and allows cached data to be used even though cache consistency can not be guaranteed. File and directory modifications are

also handled optimistically: updates are reflected in the disconnected cache and logged for later propagation to the file server when the decision is made to return to connected operation. Conflict due to overlapping updates while disconnected is possible, but rare.

Fetch-only mode is similar to disconnected mode, but differs in that it processes cache misses by requesting the needed data from the server. We use fetch-only mode frequently, both at home and when traveling, to bring missing files to a client without the cost of a full replay.

When a network is available, the user may choose to return to connected operation. The cache manger replays the log of deferred operations by iterating through the operations and propagating the modifications to the server. Before any operation is replayed, the cache manager examines server state to make sure someone else's newly created data is not destroyed. Manual error recovery is invoked if such a conflict occurs.

3. Related work

Our work with disconnected operation is inspired by the CODA project, which introduced the concept of disconnected operation and identified its usefulness for mobility [11]. CODA researchers are working on support for low bandwidth networks, such as predictive caching to obviate network demands caused by cache misses, and trickle discharging, which shares our goal of using network connectivity opportunistically without interfering with other traffic [3].

The Echo distributed file system is similar to ours in its use of write behind to reduce the latencies of operations and improve performance [14]. We depart from the Echo approach in two important ways. The first is failure semantics. We log synchronously, so when an operation completes, its changes are committed to the log and will eventually be replayed. Echo applications must either call `fsync` or a special operation that guarantees the order in which operations are committed to the server.

Echo enforces single system UNIX semantics by demanding delayed updates from client machines. In the mobile environment this requirement might be expensive or impossible to honor and can project the bandwidth latencies of mobile networks onto users of a high speed network.

4. Partially connected operation

We now describe *partially connected operation*, a technique for mobile systems that lies between connected and disconnected operation. As in disconnected operation, all file system writes are performed locally and logged. The main differences from disconnected operation are in the way it maintains client cache coherence and processes cache misses.

In partially connected mode, as in disconnected operation, `vnode` operations that cause file modifications are processed by modifying the file cache to reflect the update and creating a log entry. In some cases the ordinary AFS cache manager delegates error checking to the server, but we need to fail invalid operations as they occur, so we modified the cache manager to perform the necessary checks locally.

In disconnected mode, the cache manager behaves as though the network were unavailable and optimistically assumes that all cached data is valid. In contrast, partially connected mode assumes the availability of some communication between the client and file servers. This lets us use AFS callbacks to offer regular AFS consistency guarantees to the partially connected client: such a client opening a file is guaranteed to see the data stored when the latest (connected) writer closed the file [10]. A partially connected client sees local modifications to files before they have been propagated to the server, as do connected clients.

Directories can be tricky. A partially connected user may insert a file in a directory, while another user inserts another entry into the directory. If the cached version of the directory is used (because it has local modifications not yet propagated to the server), the entry inserted by the other user will not be seen, so we have to fetch a fresh copy of the directory and merge in our update. We plan to address this problem in the future.

On low bandwidth networks, the user may not always want the most recent version of files. For example if any files under `/usr/X11/bin/` are modified, the user may wish to continue using the cached versions instead of incurring the cost of fetching the most recent version.† We are investigating methods of providing an interface to allow this form of selective consistency.

5. Background replay

In disconnected operation, file modifications are not propagated immediately, making it difficult to share data consistently and increasing the likelihood of a conflict during replay [12]. For partially connected operation, we want to take advantage of network availability no matter what the quality if it lets us achieve timely propagation of updates, so we implemented a

† At some point the current version should be brought into the cache, but this can be deferred to a background task that runs when the system is otherwise quiescent.

background daemon to replay the log whenever opportunities arise or at the user's option.

Two significant issues arise when replaying operations in the background. First is rational management of network resources, so that the response times for interactive and other traffic do not suffer. The second issue is the effect on optimization: we and our CODA counterparts have observed that optimization of large logs can be considerable [8, 17], vastly reducing the amount of network traffic necessary for replay. Aggressive background replay may deny us this savings.

5.1. Priority queuing

The network is a primary resource in the mobile environment, so it is vital to keep replay traffic from interfering with a user's other work. Competition among various types of network traffic can increase interactive response time by causing interactive traffic to be queued behind replay traffic. Studies have shown that interactive response time is important to a user's satisfaction [18].

Similarly, replay traffic might compete with an AFS `fetch`, which is undesirable if a user is waiting for the completion of the associated read request. No user process blocks awaiting replay, so replay operations should be secondary to all other network requests.

One solution is to replay operations when the network is otherwise idle. In practice this solution is hard to implement; it is difficult to tell when a network (or other resource) is idle [4]. Furthermore, some operations, such as `store` requests, may take several minutes to complete. To avoid interference with interactive traffic, the replay daemon would need to predict a user's future behavior.

Our solution is to augment the priority queuing in our network driver. Our approach is an extension of Jacobson's compressed SLIP [9] implementation, which uses two levels of queuing in the SLIP driver: one for interactive traffic, and one for all other traffic. When the driver receives a packet for transmission, it examines the destination port to determine which queue to use. When ready to transmit a packet, it first transmits any packets on the interactive queue. The low priority queue is drained only when the interactive queue is empty.

We extend this approach by using three levels of queuing: interactive traffic, other network traffic, and replay traffic. AFS `fetch` requests are put on the second queue because a user is likely to be waiting for the completion of a read request.

When determining which packet to transmit we depart from the Jacobson method. In his SLIP implementation, the packet with the highest priority is always sent first, which for our purposes might lead to starvation of the low priority queue(s). For example, suppose the replay daemon is storing a file in the background and the user starts a large FTP PUT. FTP packets takes precedence over replay traffic, so no replay traffic will be transmitted during the duration of the FTP transfer. If the FTP transfer lasts long enough, the AFS connection will time out, and lose any progress it has made on the operation being replayed.

To prioritize the queues without causing starvation, we need a sophisticated scheduler that guarantees a minimum level of service to all traffic types. We use lottery scheduling, which offers probabilistic guarantees of fairness and service [19].

Lottery scheduling works by giving a number of lottery tickets to each item that wants to access a shared resource. When it is time to choose which item gets use of the resource, a drawing is held. The item holding the winning ticket gets access to the resource. This give a probabilistic division of the access to the resource based on the number of tickets that each item holds.

In our driver, we assign a number of tickets to each of the queues, according to the level of service deemed appropriate. When it is time to transmit a packet we hold a drawing to determine which queue to transmit from. Ticket allocation is a flexible way to configure the system and provides an easy-to-understand ''knob'' to turn for system tuning.

For the measurements described in this paper, we gave eight tickets to the interactive queue, three to the demand network traffic queue, and one to the replay queue. In future work, we plan to measure the effect of varying these static assignments. Under some circumstances, we may elect to vary them dynamically.

5.2. Delayed writes

Effective crash recovery is critical whenever delaying writes. We commit all file and metadata modifications to the log as quickly as possible, so that we don't have any dirty data in the buffer cache in a crash. Log replay works after a client crash — in our prototypes, we rely on it often (sadly).

Distinct connected clients that sequentially write a shared file don't experience a conflict, but delaying one of the updates can produce a concurrent write sharing conflict when replaying the log, so it is to our advantage to replay the log without much delay. In addition, delaying update for too long hampers the timeliness and potential usefulness of shared data; there is good reason to propagate logged operations aggressively.

On the other hand, delaying replay offers an opportunity for optimizing the log. Ousterhout reported that most UNIX files have a lifetime under three minutes and that 30–40% of modified file data is overwritten within three minutes [16]. Using our optimizer [8], we find it typical for 70% of the operations in a large log to be eliminated. It is clear that delaying log replay can significantly reduce the amount of data propagated to the server.

We may wish to enforce a minimum delay before replaying an operation, especially on networks with a per-packet cost, so that optimization could have an effect. On the other hand, if the network is available and idle and costs nothing to use, then there is nothing to be saved. On our dialups or our Ethernet, we propagate changes aggressively, whenever surplus network bandwidth is available. We run the optimizer only when changing from disconnected to connected or partially connected operation. In the future, we plan to experiment with different approaches to configuring the delay according to the varying network characteristics.

6. Results

To see how well partially connected operation performs in the low-speed and intermittent networks that interest us, we measure running times for several benchmarks. We start with hot data and attribute caches, so that comparisons between Ethernet and low-speed networks are meaningful. Later we examine the effect of a cold attribute cache.

We ran the benchmarks over Ethernet, over SLIP in connected mode (C-SLIP), and over SLIP in partially connected mode (P-SLIP). Measurements were made on a 33 Mhz Intel 486 client running Mach 2.5. We used SLIP on a 57.6 Kbps null modem connection to avoid the variability in network transfer time due to modem latencies and compression.

6.1. nhfsstone benchmark

To measure the fine-grained effect of partially connected operation on individual operations, we ran the `nhfsstone` benchmark [13], modified to remove NFS dependencies. The results in Table 1 show that P-SLIP runs substantially faster than C-SLIP, as we would expect.

Operations that run slowly over Ethernet run faster in P-SLIP, which has the advantage of deferring network requests. Because all P-SLIP operations involve synchronous logging, there is a lower bound to the running time for any particular operation.

Operation	Ethernet	C-SLIP	P-SLIP
`setattr`	21	516	33
`write`	255	3,517	123
`create`	218	2,036	99
`remove`	62	629	41
`rename`	23	294	41
`link`	36	319	40
`symlink`	125	383	47
`mkdir`	129	635	73
`rmdir`	132	351	40

Table 1. Comparison of mutating operation completion times. This table compares the time to perform various vnode operations in three cases: over an Ethernet, over SLIP in connected mode, and over SLIP in partially connected mode. The measurements were made using a version of `nhfsstone`. All times are in ms.

6.2. Andrew benchmark

We ran the Andrew benchmark [5], a synthetic workload that copies a file hierarchy, examines the copied files, and compiles source files. This benchmark, summarized in Table 2, shows that partially connected operation dramatically improves the running time of the Andrew benchmark: partially connected mode over SLIP is much faster than its connected counterpart. Because many network delays are removed from the critical path, the benchmark runs only a little slower on P-SLIP than over Ethernet.

	Ethernet	C-SLIP	P-SLIP
`MakeDir`	4	29	2
`Copy`	31	191	23
`ScanDir`	18	26	45
`ReadAll`	28	30	40
`Make`	117	507	116
`Total`	200	783	218

Table 2. Andrew benchmark results. This table shows the running time the Andrew benchmark over Ethernet in connected mode, and over SLIP in connected and partially connected mode. Both SLIP measurements started with hot data and attribute caches. All times are in seconds. Integer roundoff accounts for differences between column entries and column totals.

These examples show that partially connected mode improves the response time for file system operations. With a hot cache, a remote user can expect tasks to run almost as fast as in the office environment.

6.3. Replay time

We measured the log replay time for the Andrew benchmark. Figure 1 shows the size of the log as the Andrew benchmark runs through its phases. The solid horizontal line along the bottom of the graph shows the running times of the MakeDir, Copy, and Make phases of the benchmark. (The ScanDir and ReadAll phases are read-only.) The dashed horizontal line shows the time at which the corresponding parts of the log were replayed.

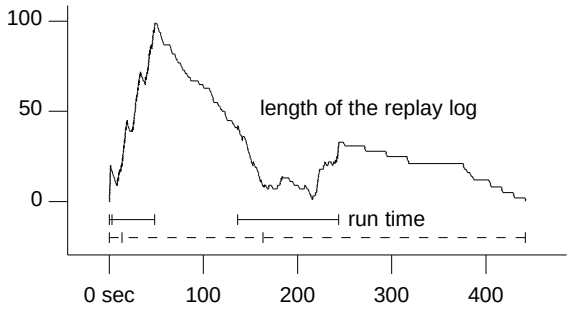

Figure 1. Length of replay log. This figure shows the number of operations in the replay log while running the Andrew benchmark over partially connected SLIP. The solid horizontal line show the running time of the MakeDir, Copy, and Make phases of the benchmark. The ScanDir and ReadAll phases are not shown, as they issue no network requests. The dashed horizontal line shows the times at which the operations logged by these three phases are run.

A total of 222 operations are logged, with no more than 99 operations pending at any time. Table 3 shows that logged operations are delayed over a minute on average, and up to four minutes in the extreme.

	average	maximum
cold cache	62	207
hot cache	87	249

Table 3. Delay time in the replay log. This table shows the average and maximum time that operations await replay while running the Andrew benchmark. When running with a cold attribute cache, the benchmark is frequently stalled, giving the replay daemon more opportunities to work on the log, resulting in shorter average and maximum delays. The total time to run the benchmark and exhaust the log is about the same in both cases. All times are in seconds.

6.4. Replay interference

The running times of the Andrew benchmark vary according to whether the data and attribute caches are hot or cold, as well as whether the replay daemon is running or idle. Table 4 shows the effect of controlling these variables.

	I	II	III	IV
cache	hot	cold	cold	hot
replay	off	off	on	on
MakeDir	3	3	2	3
Copy	19	26	47	23
ScanDir	14	15	48	45
ReadAll	25	25	40	40
Make	94	96	106	116
Total	155	165	249	218

Table 4. Cold cache Andrew benchmark results. This table shows the effect of running the Andrew benchmark with a hot or cold attribute cache, and with the replay daemon running or disabled. All times are in seconds. Integer roundoff accounts for differences between column entries and column totals.

A cold attribute cache slows the pace of the benchmark, giving the replay daemon more opportunities to whittle away at the log in the earlier phases, so that the average and maximum delay of logged operations is decreased, as shown in Table 3. The replay daemon itself causes the benchmark to slow down by 40–50% overall. This may be due in part to network contention, but even the ScanDir and ReadAll phases, which hit hot caches and don't use the network at all, run slower when background replay is active. This points the finger at contention for local resources; we suspect competition for locks in the AFS cache manager to be a contributor.

There is one mysterious entry in Table 4: the run time of the Copy phase for cold cache + active replay daemon (case III, 47 sec.). This is the hard case, where background replay has the most opportunities to interfere with interactive operations. Still, the penalty is higher than we expect, and we think there is a bug in our implementation.

To isolate the effect of network interference caused by running the replay daemon, we ran the Andrew benchmark in fetch-only mode with the replay daemon turned on and off. The run times should be comparable to the corresponding hot-cache trials in Table 4, but in fact the benchmark runs quite a bit faster in fetch-only mode, as shown in Table 5.

The difference in run-times caused by enabling the replay daemon in fetch-only mode is a little more than 10%; we believe that we will be able to debug and tune our implementation to achieve these differences in partially connected mode as well. With replay disabled, hot cache run times do not differ much for partially connected mode, fetch-only mode, and disconnected mode (not shown).

	I′	IV′	local
cache	hot	hot	——
replay	off	on	——
MakeDir	2	2	3
Copy	18	23	13
ScanDir	14	16	13
ReadAll	24	42	23
Make	93	104	88
Total	152	197	139

Table 5. Andrew benchmark results for fetch-only mode and local disk. This table shows the effect of running the Andrew benchmark in fetch-only mode with a hot attribute cache, and with the replay daemon running or disabled. All times are in seconds. Integer roundoff accounts for differences between column entries and column totals. For comparison, local disk times are also shown here.

7. Discussion

Partially connected operation promises to improve response time and reliability, while interfering only slightly with AFS cache consistency guarantees. AFS guarantees that a client opening a file sees the data stored when the most recent writer closed the file. Because a partially connected client does not immediately propagate changes, other users can not see modified data. Furthermore, conflicts may occur if partially connected users modify the same file. In our experience, these conflicts are rare; a substantial body of research concurs by showing that this kind of file sharing is rare [2, 11, 16].

If stronger guarantees are needed, they might be provided by server enhancements. For example, an enhanced consistency protocol might inform servers that dirty data is cached at a client; when another client requests the data, the server can demand the dirty data, as is done in Sprite [15] and Echo.

We choose not to implement this mechanism for several reasons. First, it assumes that the server is able to contact the client on demand, an assumption that may not always be true. Additionally, demand fetching can place a severe strain on a client's network connection. Because of the limited bandwidth, one user may see her effective bandwidth drastically reduced because another user is reading a file that she has modified; this may not be acceptable to all users. Finally, such a change would require changing all of the AFS servers in the world to support the new protocol; practically speaking, this is out of the question.

References

1. D. Bachmann, P. Honeyman, and L.B. Huston, ''The Rx Hex,'' in *Proc. of the First Intl. Workshop on Services in Distributed and Networked Environments*, Prague (June 1994).
2. Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, ''Measurements of a Distributed File System,'' in *Proc. of the 13th ACM Symp. on Operating Systems Principles*, Asilomar (October 1991).
3. Maria R. Ebling, Lily B. Mummert, and David C. Steere, ''Overcoming the Network Bottleneck in Mobile Computing,'' in *Proc. of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz (December 1994).
4. Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes, ''Idleness is Not Sloth,'' pp. 201–212 in *Proc. of the USENIX Conf.*, New Orleans (January 1995).
5. J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West, ''Scale and Performance in a Distributed File System,'' *ACM Transactions on Computer Systems* **6**(1) (February, 1988).
6. John H. Howard, ''An Overview of the Andrew File System,'' pp. 23–26 in *Proc. of the Winter USENIX Conf.*, Dallas (January 1988).
7. L.B. Huston and P. Honeyman, ''Disconnected Operation for AFS,'' in *Proc. of the 1993 Symp. on Mobile and Location-Independent Computing*, Cambridge (August 1993).
8. L.B. Huston and P. Honeyman, ''Peephole Log Optimization,'' in *Proc. of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz (December 1994).
9. V. Jacobson, ''Compressing TCP/IP Headers for Low-Speed Serial Links,'' RFC 1145, Network Information Center, SRI International, Menlo Park (February 1990).
10. Michael Leon Kazar, ''Synchronization and Caching Issues in the Andrew File System,'' in *Proc. of the Winter USENIX Conf.* (February 1988).
11. J.J. Kistler and M. Satyanarayanan, ''Disconnected Operation in the Coda File System,'' *ACM Transactions of Computer Systems* **10**(1) (February 1992).
12. James J. Kistler, ''Disconnected Operation in a Distributed File System,'' Ph.D. Thesis, Carnegie Mellon University (May 1993).
13. Legato Systems, Inc., *NHFSSTONE*, July, 1989.

14. T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart, ‘‘A Coherent Distributed File Cache with Directory Write-behind,’’ SRC Research Report #103, Digital Equipment Corporation (June 1993).

15. M. Nelson, B. Welch, and J. Ousterhout, ‘‘Caching in the Sprite Network File System,’’ *IEEE Transactions on Computers* **6**(1) (February 1988).

16. J. Ousterhout, H.L. DaCosta, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, ‘‘A Trace-Driven Analysis of the Unix 4.2 BSD File System,’’ in *Proc. of the 10th ACM Symp. on Operating Systems Principles*, Orcas Island, WA (December 1985).

17. M. Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Pumeet Kumar, and Qi Lu, ‘‘Experience with Disconnected Operation in a Mobile Computing Environment,’’ in *Proc. of the 1993 Symp. on Mobile and Location-Independent Computing*, Cambridge (August 1993).

18. B. Shneiderman, *Designing the User Interface*, Addison-Wesley (1987).

19. Carl A. Waldspurger and William E. Weihl, ‘‘Lottery scheduling: flexible proportional-share resource management,’’ pp. 1–11 in *Proc. First Symp. on Op. Sys. Design and Impl. (OSDI)*, Monterey (Nov. 1994).

Availability

Researchers with an armful of source licenses may contact `info@citi.umich.edu` to request access to our AFS modifications.