Partially Connected Operation

L.B. Huston
lhuston@citi.umich.edu

P. Honeyman
honey@citi.umich.edu

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 improves matters, 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 [8, 11], a form of optimistic replication that allows the use of 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 are often 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: by treating 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 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 appropriate for low bandwidth networking is to exploit cache hits aggressively, but to fetch files when cache misses occur. This *fetch-only* mode of operation does not support the AFS cache consistency guarantees, 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, so the chance that two users might concurrently update the same file grows.

Lying among connected, disconnected, and fetch-only operation is a mode of operation that allows us to hide many of the network latencies, yet 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 benchmark measurements that illustrate the effectiveness of the technique.

A preliminary version of this paper appears in *Proc. of the Second USENIX Symposium on Mobile and Location-Independent Computing*, Ann Arbor (April 1995).

2. Background

The work presented in this paper is based on a version of the AFS client that supports disconnected operation [8]. 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 propagation to the file server when the decision is made to return to connected operation. Conflict due to overlapping updates is rare and generally benign.

Fetch-only mode differs from disconnected mode in the way that it processes cache misses. Where disconnected mode aborts any request that can't be satisfied out of the cache, fetch-only mode requests and caches the data from the server. We use fetch-only mode frequently, 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 triggered 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 resembles ours in its use of write behind to reduce the latencies of operations and improve performance [14]. We depart from Echo in two important ways. The first is the failure semantics associated with synchronous logging. When an operation completes, its changes are committed to the log. This provides strong assurance that they will eventually be replayed. Echo applications that need durability guarantees must either call `fsync` or issue a special operation that adjusts the order in which operations are committed to the server.

The second difference is the semantics of delayed writes. An Echo server issues a demand for delayed updates from a client machine when single system UNIX semantics require it. 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 file 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 we maintain client cache coherence and process 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: a client opening a file is guaranteed to see the data stored when the latest (connected) writer closed the file [10]. Of course, all AFS clients, including partially connected ones, see their local modifications before the file is closed and updates are propagated to the server.

Directories are tricky. If a partially connected user inserts a file in a directory, and another user later inserts another file, it is difficult to make both updates visible on the partially connected client. If the cached version of the directory is used, modifications by other users can not be seen until replay. Similarly, importing the updated directory from the server makes the local modifications invisible until replay. Merging the two versions requires a partial replay of the log, which can consume an arbitrary amount of time. In our implementation, we use the locally updated directory, in analogy to viewing updates in an open file.

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. The Odyssey system [16] has provisions for adapting file system behavior to network conditions, and might be useful here.

5. Background replay

In disconnected operation, file modifications are not propagated immediately, which makes it inconvenient to share data and increases 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 a consistent cache and timely propagation of updates, so we implemented a background daemon to replay the log whenever opportunities arise (or at the user's discretion).

Two significant issues arise when replaying operations in the background. The first is rational management of network resources, to prevent response times for interactive and other traffic from suffering. The second issue is the effect on optimization: we and our CODA counterparts have observed that optimization of large logs can be considerable [7, 18], 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. Studies have shown that interactive response time is important to a user's satisfaction [19]. Competition among various types of network traffic can increase interactive response time if interactive packets are queued behind replay traffic.

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 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 the 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. When determining which packet to transmit we depart from Jacobson. 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 queues.

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 scheduler that guarantees a minimum level of service to all traffic types. We use *lottery scheduling* [20], which offers probabilistic guarantees of fairness and service.

Lottery scheduling works by giving a number of lottery tickets to each item that wants access to a shared resource. When it is time to allocate the resource, a drawing is held. The item holding the winning ticket gets access to the resource. This gives a probabilistic division of 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. We would expect relative throughput to vary along with ticket allocation, as Table 1 confirms.

Replay tickets	Replay throughput	FTP throughput
0	0.0	1.0
16	0.16	0.83
20	0.12	0.90
24	0.26	0.73
31	0.30	0.72
50	0.43	0.53
100	1.0	0.0

Table 1. Varying lottery ticket ratio. This table shows the effect on throughput as the allocation of lottery tickets is varied. Maximum throughput for a given queue is achieved when all tickets are allocated to that queue. 100 tickets are used.

We primed the replay log with numerous 200 KB store operations, then initiated a 10 MB FTP `put` while simultaneously replaying the log. We allocate no tickets to interactive traffic, and 100 tickets to the remaining traffic. The replay and FTP throughputs, shown as fractions of the maximum achievable, match fairly closely their respective ticket allocations.

Allowing replay traffic to compete for access to the network can adversely affect interactive response time. To gauge the impact, we primed the client with a large replay log and measured TELNET round trip times with `telnetping`, an application from Morning Star Technologies that sends a slow stream of TELNET packets to a remote server and measures the response time. Table 2 shows how round-trip time varies as tickets are reallocated.

Even when interactive traffic is given the lion’s share of tickets, other packets are scheduled for transmission between the `telnetping` packets. Thus `telnetping` packets arrive at the scheduler to find a replay packet in the process of being transmitted. The scheduler is not preemptive, so a queueing time averaging one half of a packet transmission time delays the interactive packet. `Telnetping` measures round trip delay, so we expect the total round-trip delay to be one packet transmission time. For our 576-byte packets and our 38.4 Kbps SLIP line, that amounts to 150 msec. Table 2 confirms that interactive traffic is delayed by about this amount, and that the delay increases as fewer tickets are given to interactive traffic.

Interactive tickets	Round-trip delay	Standard deviation
100	11	2
80	168	193
75	181	259
66	195	98
40	301	294

Table 2. Interactive response time. This table shows how lottery scheduling in the network driver affects interactive response time. Interactive packets are delayed because the scheduler is not preemptive. Times are in msec. 100 tickets are used.

5.2. Delayed writes

Effective crash recovery is critical when writes are delayed. We commit all file and metadata modifications to the log synchronously with the request 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 still rely on it occasionally.

Distinct connected clients that sequentially write a shared file don’t experience a conflict, but delaying the propagation of one or both updates can produce a concurrent write sharing conflict when the log is replayed, so it is to our advantage to replay the log without much delay. In addition, delaying update reduces the timeliness and potential usefulness of shared data. In short, there is ample motivation to propagate updates aggressively.

On the other hand, delaying replay offers an opportunity for optimizing the log. Ousterhout reports that most UNIX files have a lifetime under three minutes and that 30–40% of modified file data is overwritten within three minutes [17]. Using our optimizer [7], we find it typical for 70% of the operations in a large log to be eliminated. In fact, the larger the log, the greater the fraction of operations eliminated by the optimizer. It is clear that delaying log replay can help 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. Benchmarks

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. For the measurements described in this section, we gave eight tickets to the interactive queue, three to the demand network traffic queue, and one to the replay queue.

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 38.4 Kbps null modem connection for our tests. A direct connect avoids the potential for variability in network transfer time caused by modem latencies and compression.

6.1. `nhfsstone` benchmark

To measure the fine-grained effect of partially connected operation on individual operations, we modified the `nhfsstone` benchmark [13] to remove NFS dependencies. The results in Table 3 show that P-SLIP runs substantially faster than C-SLIP, as we would expect.

Because all P-SLIP operations involve synchronous logging, there is a lower bound to the running time for any particular operation. The session semantics of AFS make it difficult to interpret `write` times, so we omit them here. In our tests, though, partially connected `store` operations ran in about the same amount of time as the other operations, bounded by synchronous logging to the local disk, while store operations over either network took considerably longer.

Operation	Ethernet	C-SLIP	P-SLIP
`setattr`	12.8	99.3	52.9
`create`	18.9	101.6	59.8
`remove`	39.5	171.5	64.8
`rename`	26.8	183.6	71.3
`link`	26.2	129.8	71.4
`symlink`	38.3	168.7	66.3
`mkdir`	99.8	286.1	76.6
`rmdir`	37.2	99.4	60.4

Table 3. 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 milliseconds.

6.2. Andrew benchmark

We ran the Andrew benchmark [5], a synthetic workload that copies a file hierarchy, examines the copied files, and compiles them. We find 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. Table 4 details the running time of the benchmark.

Phase	Ethernet	C-SLIP	P-SLIP
`MakeDir`	1	12	2
`Copy`	16	144	20
`ScanDir`	15	18	15
`ReadAll`	26	24	26
`Make`	67	385	70
`Total`	125	583	133

Table 4. Andrew benchmark results. This table shows the running time of the Andrew benchmark over Ethernet in connected mode, and over SLIP in connected and partially connected mode. All measurements start with hot data and attribute caches. All times are in seconds.

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

The time to run the Andrew benchmark and exhaust the log is about the same as the time to run the benchmark in connected mode over SLIP. Figure 1 shows the size of the log as the benchmark runs through its phases. A total of 222 operations are logged, with up to 129 operations pending at one time. 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.

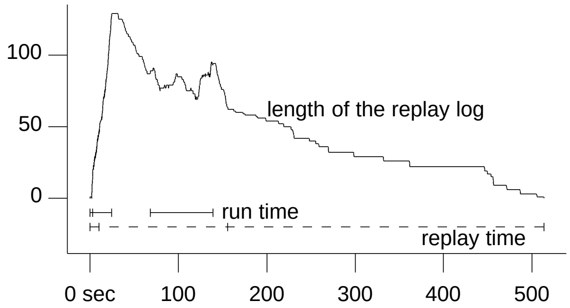

Figure 1. Length of replay log. This figure shows the number of operations in the replay log while running the Andrew benchmark over partially connected SLIP. The solid horizontal line show the running time of the MakeDir, Copy, and Make phases of the benchmark. The ScanDir and ReadAll phases are not 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.

Table 5 shows that logged operations are delayed over a minute on average, and up to six minutes in the extreme. Because the Andrew benchmark does not account for idle or ‘‘think’’ time, the log grows more rapidly than we would expect in practice. Furthermore, these idle periods would make the replay daemon more effective, so that in real use we would expect the average and maximum log size to be shorter than is reflected here.

6.4. Replay interference

The running time of the Andrew benchmark depends on whether the data and attribute caches are hot or cold, as well as whether the replay daemon is running or idle. When the replay daemon is disabled, hot cache run times do not differ much for partially connected mode, fetch-only mode, and disconnected mode. 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 we saw in Table 5.

	average	maximum
cold cache	88	314
hot cache	107	374

Table 5. 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. All times are in seconds.

Table 6 shows that total run time of the benchmark increases by 5–15% when the replay daemon is running. Most of the slowdown is in the CPU-intensive Make phase.

	cold cache		hot cache	
replay:	off	on	off	on
`MakeDir`	2	2	2	2
`Copy`	30	36	20	20
`ScanDir`	13	14	13	15
`ReadAll`	25	27	25	26
`Make`	71	80	65	70
`Total`	141	159	125	133

Table 6. 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.

7. Discussion

Partially connected operation improves performance, response time, and reliability, while interfering only a little 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, 17].

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.

Experience with partially connected operation has been positive so far. While we designed with remote access to AFS in mind, partially connected operation can play an important role in the office environment as well, where it can offer performance and availability advantages over connected AFS.

References

1. D. Bachmann, P. Honeyman, and L.B. Huston, ''The Rx Hex,'' in *Proc. of the First Intl. Workshop on Services in Dist. and Networked Env.*, 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 SOSP*, 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 Comp. Sys. and Appl.*, 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 TOCS* **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, ''Peephole Log Optimization,'' pp. 25–32 in *Bulletin of the Tech. Comm. on Op. Sys. and Appl. Env.*, IEEE Computer Society (Spring 1995).
8. L.B. Huston and P. Honeyman, ''Disconnected Operation for AFS,'' in *Proc. of the USENIX Symp. on Mobile and Location-Independent Comp.*, Cambridge (August 1993).
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 TOCS* **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 Trans. on Comp.* **6**(1) (February 1988).
16. Brian Noble, M. Satyanarayanan, and Morgan Price, ''A Programming Interface for Application-Aware Adaptation in Mobile Computing,'' *this journal*, pp. ??–?? (??).
17. 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 SOSP*, Orcas Island, WA (December 1985).
18. 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 USENIX Symp. on Mobile and Location-Independent Comp.*, Cambridge (August 1993).

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

20. 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).

Larry Huston holds B.S.E. and M.S.E. degrees in Aerospace Engineering and Computer Engineering and the Ph.D. in Electrical Engineering and Computer Science, all from the University of Michigan. His dissertation is titled *Remote Access to Distributed File Systems.* He is employed by Ipsilon Networks, Inc.

Peter Honeyman holds the B.G.S. (with distinction) from the University of Michigan. and the Ph.D. from Princeton University for research in relational database theory. He has been a Member of Technical Staff at Bell Labs and Assistant Professor of Computer Science at Princeton University. He is Associate Research Scientist at the University of Michigan's Center for Information Technology Integration and Adjunct Associate Professor of Electrical Engineering and Computer Science.

Honeyman has been instrumental in several software projects, including Honey DanBer UUCP, PathAlias, MacNFS, Telebit's UUCP spoof, and improvements to Transarc's AFS, such as transport layer improvements and disconnected operation. His research efforts focus on distributed file systems, with an emphasis on mobile computing, security, and performance.

Availability

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