CITI Technical Report 93–3

Disconnected Operation for AFS

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

P. Honeyman
`honey@citi.umich.edu`

ABSTRACT

AFS plays a prominent role in our plans for a mobile workstation. The AFS client manages a cache of the most recently used files and directories. But even when the cache is hot, access to cached data frequently involves some communication with one or more file servers to maintain consistency guarantees. Without network access, cached data is soon rendered unavailable.

We have modified the AFS cache manager to offer optimistic consistency guarantees when it can not communicate with a file server. When the client reëstablishes a connection with the file server, it tries to propagate all file modifications to the server. If conflicts are detected, the replay agent notifies the user that manual resolution is needed.

Our system brings the benefits of contemporary distributed computing environments to mobile laptops, offering a fresh look at the potential for nomadic computing.

June 18, 1993

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

Disconnected Operation for AFS

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

P. Honeyman
honey@citi.umich.edu

1. Introduction

The continuing miniaturization of computer hardware hath wrought dramatic changes in portable computers. In tandem, the ties to wall jacks and other wiring requirements are being unbound, which has led to an explosion of interest and activity in nomadic computing. Yet, operating system support for mobile computers has not kept pace, as the research and commercial computing communities have embraced the distributed computing paradigm, in which network connectivity forms a fundamental technological underpinning. To close the gap between these advances in hardware and software, the Center for Information Technology Integration (CITI) has sponsored the LITTLE WORK project [1], which is investigating the operating system requirements for nomadic computers.

The goal of the LITTLE WORK project is to build a mobile computing platform that closely resembles CITI's desktop computing environment. Our current prototype is an Intel i386-based laptop computer running Mach 2.5 from Carnegie Mellon University [2], along with MIT's X Window System [3]. When traveling, we use modems and SLIP [4] to run conventional network-based services such as TCP/UDP/IP, NTP [5], Kerberos [6], and AFS [7].

A key component of our mobile workstation is the distributed file system. There are numerous benefits to using a caching distributed file system instead of a standalone file system on a mobile machine. In the latter case, a user preparing for a trip with her laptop must select the files she will need and manually copy them to the laptop. On her return, she must copy any modified files back to permanent storage, usually a file server or the local disk on her desktop machine.

Compare this to a caching distributed file system. To prepare for a trip, the user attaches the laptop to the network and runs the applications that she intends to use while traveling. This causes the caching mechanism to copy the latest version of the referenced data to the local disk if it is not already there. After heating up the cache, she disconnects from the network and hits the road. Upon arriving at a location that supports network connectivity, possibly her home base, she establishes a network connection and directs the file system to propagate her modifications to the file server.

The first benefit of the distributed file system approach is that it reduces the potential for human error. The cache manager logs all file modifications, so the user need not worry about forgetting to copy files to and from permanent storage. Furthermore, the distributed file system offers the potential to detect conflicts that arise if shared files are modified by more than one party, resolves those conflicts that it can, and reports all others.

Another advantage is that a distributed file system adapts as a user's work habits change. Laptops tend to be constrained in disk space, limiting the number of applications that can be installed. In a traditional system, when the user wants to use a different application, she must manually install a new program. As part of this installation process she may need to make space for the new application by removing some other files. She may later regret her selection. In a distributed file system, applications are installed by system administrators and are accessed from any machine. The cache manager takes care of copying the necessary files to the local disk, as well as removing files that haven't been used for a long time.

For our distributed file system we use AFS from Transarc. While AFS works well in a desktop environment, it fails completely in a partitioned network. The difficulty is that AFS consistency guarantees require the client cache manager to maintain network connectivity with the servers responsible for data in the client cache. When a server gives some data to a client, the server also issues a ‘‘callback.’’ This callback is a promise that the server will notify the client if the cached file is modified. The client uses possession of this callback to ensure that the cached version of the data is the most recent. If a network partition disrupts communication between the client and the file server, the cache manager can not be sure that the server has not tried to revoke any callbacks. The cache manager assumes the worst case: all cached data becomes invalid. At this point, the cache manager refuses to allow access to any of its cached data, even though the preponderance of the data is valid.

Mobile computers enjoy only sporadic network connectivity, and require AFS to be more resilient to network partitioning. The cache manager must allow access to cached copies of files and directories, performing the necessary consistency checks only when a network connection is established with the file server.

2. Related Work

Previous work on disconnected access to a distributed file system includes the Coda project at Carnegie Mellon University [8, 9, 10]. Coda is a distributed file system similar to AFS, with additional support for server replication. Voluntary disconnection by a client is treated as a special case of network partitioning. A disconnected client can continue working by using any data is has in its cache. When the client reconnects to a network, it gives the collection of updates made while disconnected to an agent that reintegrates the updates into the file system using methods similar to those used to resolve updates across replicated servers. Because Coda enlists support from the file server in reintegrating client updates, it can offer strict transactional guarantees on the entire collection of updates.

UCLA’s Ficus replicated file system [11] also supports a form of disconnected access. Ficus uses peer-to-peer operations on replicated files instead of second-class replication (*i.e.,* caching) by clients of a centralized file server. As a replicated server, a mobile Ficus workstation can achieve many of the goals of disconnected operation.

Although Coda and Ficus provide for service while disconnected, we have elected to go our own way with AFS, for several reasons. Most importantly, AFS is where our files are; it would not make sense to use a file system that we don’t use every day. We have been satisfied AFS users for many years, and we understand its behavior, so our choice is to adapt AFS for disconnected operation. A related reason to stay with AFS is that there are currently over 75 different cells[1] accessible from our workstations. If we switched to another file system, we would lose the ability to use these cells while disconnected. In short, were we to go with another file system, we would lose access to our own and CITI’s resources and the home directories of our colleagues.

3. Design

Before modifying AFS, we developed some basic design criteria. First, we prohibit any changes to the AFS servers, restricting our effort to the client cache manager. If our modifications included any changes to the file servers, we would lose the ability to access other AFS cells. The decision to make the client solely responsible for disconnected operation is significantly different from the approach taken by Ficus and Coda. We feel that the benefit of continued access to AFS outweighs potential problems.

Our second design rule is that we are concerned only with disconnected operation. Coda and Ficus provide a high degree of availability by replicating servers, as well as allowing systems to use local data when a file server is not reachable. But disconnected clients exist in a network partition containing a single machine; replicated servers do not provide an advantage for nomadic computing.

This leaves as the primary issue how to modify the cache manager. AFS tries to provide strict consistency guarantees, to wit, AFS guarantees that a client opening a file sees the data stored when the the most recent writer closed the file. This guarantee is hard to honor in a partitioned network. The main difficulty arises because AFS is pessimistic. In the absence of firm evidence to the contrary, the cache manager assumes that cached data is invalid. Our approach is to modify the cache manager to be more optimistic, allowing access to cached files. This needs to be done

[1] A *cell* is an AFS administrative boundary, comparable to a Kerberos *realm.* Some examples of cells are `umich.edu`, `citi.umich.edu`, `alw.nih.gov`, and `cern.ch`.

with care. While it is acceptable to create an inconsistent view of the file system in the cache of the mobile machine, we don't want to store any data back to the file server that will violate the AFS consistency guarantees.

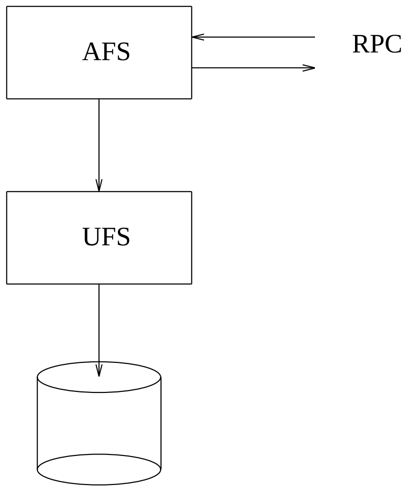
The archetypal conflict occurs when an object modified by a connected client is also modified by a disconnected client. If the disconnected client blindly stores data back to the file server, then the other client's modifications will be overwritten. In practice these types of conflicts are rare; for example, Ousterhout *et al.*, showed that under work loads similar to ours, write sharing rarely occurs [12], and later studies agree that in such environments, write sharing remains rare [9, 13]. Our optimism has also been validated by running simulations using traces gathered from file servers that we access on a daily basis [14].

4. Running Disconnected

To make disconnected AFS work, we made several changes to the cache manager. One change is to make the cache manager optimistic about cache consistency. We elected to make these modifications at the vnode layer [15]. There are several reasons for this decision. First, it is easier to determine the ''correct'' behavior because we have access to all the cache manager's information.

In contrast, consider the option of masquerading as a file server at the remote procedure call (RPC) layer. The response to an RPC depends on the the current contents of the cache, as well as other data structures maintained by the cache manager. For example, suppose a user is trying to access a file for which the cache manager no longer has a callback. Normally the cache manager would issue a `getstatus` RPC to the file server, which returns a callback and the current version number of the file. While disconnected, it is not possible to issue this RPC, so we would like the cache manager to assume (optimistically) that its cached data are valid. From the RPC layer, this is difficult because the response must be based on the current contents of the cache. If the cache manager doesn't have a cached copy of the file's status, then the request must fail. But if the file's status is cached, then the `getstatus` request should return the cached information. In either case the RPC layer needs to have knowledge that is easily accessible at the vnode layer.

Another reason to make the modifications at the vnode layer is to aid in propagating file modifications back to the file server. Operations to AFS files consist of a pair of vnode operations: local ones, or `ufs_vnodeops`, and remote ones, or `afs_vnodeops`. When an operation is applied to an AFS file, the appropriate vnode operations are called. The `afs_vnodeop` checks cache validity, fetches current versions of the files, stores any modifications back to the file server, *etc*. Communication with file servers is through the RPC layer.



AFS vnode architecture. The AFS vnode operations use the RPC layer to communicate with the file server, then call the underlying UFS vnode operations to access the data in the local disk cache.

After the AFS portion of the vnode operation is completed, `ufs_vnodeops` are called on to manipulate the cached data. Consider a `read` operation: the AFS vnode operation ensures that the cached copy of the data is valid, whereupon the UFS vnode operation, `ufs_read`, is called on to return the data out of the local cache.

When the cache manager is disconnected, `ufs_vnodeops` are performed while `afs_vnodeops` are logged and deferred. When a network connection is established, the cache manager iterates through the log of deferred `afs_vnodeops` and tries to replay the operations to the file server. By logging at the vnode layer, we are assured that after replaying the log, all of the `ufs_vnodeops` and `afs_vnodeops` have been performed. In the absence of any conflicts, we will see the same final state that would have resulted had the operations been performed while connected.

This creates a good basis for the replay algorithm, but we need to modify the algorithm to account for operations that might violate the AFS consistency guarantees. This issue is discussed in the next section.

To put the cache manager into disconnected mode, the user issues a `disconnect` command. Thereafter, every successful vnode operation on a file generates a corresponding log entry; operations that fail are not logged. Along with the type of operation, enough extra information is logged to allow the replay agent to execute the same operation when reconnecting with the file server. The extra information depends on the type of operation, but is typically such data structures as file name, AFS' internal file identifier, and current data version number. If the operation is one that modifies the state of the file system, the files in the local cache are modified to reflect the change.

The principal difference in the cache manager's behavior when running disconnected is the way it enforces consistency. Ordinarily, before the cache manager references an object, it assures that it has a callback for that object; if not, then the cache manager issues a `getstatus` RPC to the server to get a callback. When the cache manager is disconnected, it can not perform this or any other RPC. Instead it marks the object as having a callback issued by the local host. (Normally a callback is marked with the IP address of the server responsible for the object.) The locally issued callback is used until AFS is reconnected.

In disconnected mode, the occasional cache miss is an inevitable fact of life. Under normal circumstances, the cache manager asks a file server for the missing information. As a disconnected cache manager can not get this information, it returns an appropriate error (`ENETDOWN`) to the calling program.

In our travels with a LITTLE WORK along, we occasionally find that we are missing an important file, yet we are not willing to pay the price of replaying a substantial log, so we have added a ‘‘fetch-only’’ mode of disconnected operation. In fetch-only mode, the cache manager issues `afs_vnodeop` RPCs for non-mutating operations, and logs mutating `afs_vnodeop`s.

Although early versions of AFS cached whole files, the current version breaks large files into 64 KB chunks. This chunking lets an AFS client work with files larger than the local disk. This has scant advantage for a mobile computer, and poses potential problems should we discover that an essential file, say `/usr/X11/bin/X386`, is only partially resident in the cache. Our inclination is to revert to whole-file caching; as an expedient (read *hack*), we set the chunk size on our mobile clients to be one megabyte.

Another problem we have encountered is applications that unnecessarily demand more information than the disconnected cache manager can provide. A potent example is the UNIX file removal program, `rm`. In this code fragment, taken from the Berkeley UNIX `rm.c`, the program obtains status information about the target to be removed, so that it can complain about a request to remove a directory or a protected file.

```
if (lstat(arg, &buf)) {
   if (!fflg) {
      fprintf(stderr, ...);
      errcode++;
   }
   return (0);   /* error */
}
```

If we try to remove a file for which we don't have status information cached, `rm` fails. (Observe that the `−f` flag does nothing more than make `rm` fail silently, in this case.) Yet the underlying vnode operations are capable of performing the operations required to remove the file, although there are complications on replay. We have modified our copy of `rm.c`, but we're not happy about it.

5. Replaying the Log

After running disconnected for a while, a network connection must be established to propagate modifications back to the file server. Once a connection is established, the user issues the `reconnect` command. This command iterates through the log of deferred `afs_vnodeops` and attempts to perform all the delayed operations.

A problem arises when the same file is modified by a connected client and a disconnected client. Before replaying an operation that changes the state of the file server, say a `storedata` RPC, the replay agent checks to see whether the data has been modified by another client. If so, we consider the case to be an instance of concurrent write sharing.[2] We don't have tools to resolve such a conflict, so the replay agent is left with the

[2] We don't assume strictly synchronized clocks, so we can not use file modification timestamps to discriminate sequential write conflicts from concurrent ones. Otherwise, we might consider the traditional UNIX approach: ‘‘last writer wins.’’

responsibility for preserving both the modified data on the file server and the modified data in its cache. Our solution is to store the locally modified data on the file server in a renamed object, so that both versions are stored on the file server. The replay agent also lets the user know a conflict has occurred, and encourages her to resolve the problem by hand.

AFS vnode operations can be divided into two classes. The first class is the non-mutating operations: these operations do not cause any changes to file server state. An example of this type of operation is `afs_read`. Non-mutating operations can not be involved in write conflicts, and don't require information transfer to the file server at replay. Nonetheless, we log these operations because some useful information can be determined from them, *e.g.*, we can detect when stale data may have been used.

The second class of operations are the mutating ones, those that modify the file server's state. An example of a mutating operation is `afs_create`. Mutating operations pose the most concern, as they have the potential to be involved in write conflicts.

In the next sections, vnode operations of both classes are listed along with the appropriate methods for performing the replay and detecting conflict. Resolving data conflicts is left to the user, but in many cases, the replay manager can resolve directory conflicts on its own.

5.1. Non-mutating operations

Non-mutating operations are logged to help determine if stale data was used. Non-mutating operations do not cause any changes on the file servers, and can not be involved in write conflicts. However, it is possible for non-mutating operations to be performed on stale data. The goal in replaying non-mutating operations is to detect any stale data usage so that the user can be warned appropriately.

Some of the algorithms below use timestamping to compare the modification times of files with the time that the operation was performed. This can be troublesome because a disconnected client has no means of running a time synchronization protocol. Consequently we need to rely on the laptop clock not drifting too far. To help account for drift we consider changes within a time window instead of a single instance in time.

5.1.1. `afs_open`

To determine if a file was stale when it was opened, the replay agent fetches the current version number of the file from the server and compares it to the version number of the file when it was cached. If the version numbers are identical, the file was valid when used locally. If they differ, stale data may have been used; this depends on whether the `open` preceded the modification of the file.

Because an AFS server increments file version numbers by one each time a file is modified, the replay agent can tell how many times the file was modified on the server. If the file was modified more than once, it is impossible to know the earliest modification time, as only the latest modification time is stored. If the file was modified exactly once, a comparison of the modification time with the local open time determines whether stale data was used. If the file was modified more than once, and the local open time does not precede the modification time, it is possible that stale data was used, but we can't know.

Using these rules, the replay agent reports to the user whether stale data was used or may have been used.

5.1.2. `afs_read`

The rules for determining the use of stale data are applied at the time the file is opened, not when it is read, so no warnings are issued here.

5.1.3. `afs_lookup`

It is possible to use the same algorithm used by `afs_open` to determine whether stale data was used during a lookup. However, this information may not provide any useful information to users, and because of their high frequency, logging potential lookup conflicts might generate so much output that it would obscure important messages from the replay agent. Therefore, we do not report stale lookups, but we do record them for our own analysis.

5.1.4. `afs_getattr`

Attribute modification times are not exported via AFS, so the replay agent can not know when the attributes were changed. As a heuristic to determine whether `afs_getattr` used stale data, the replay agent compares the current attributes with the attributes of the cached file. If the attributes have changed since the last time they were cached, the replay agent informs the user that stale attributes may have been used.

5.1.5. `afs_readlink`, `afs_access`, `afs_readdir`

The issues for these vnode operations are similar to `afs_lookup`. It is possible to determine if stale data was used, but this does not seem to provide any useful information for the user. Therefore, we treat these operations as we do `afs_lookup`.

5.2. Mutating operations

The mutating vnode operations offer the most challenge. These operations modify server data, so they have the potential to be involved in write conflicts and violations of AFS guarantees.

Most conflicts are resolved by creating a new instance of the object, and dumping the contents of the disconnected version into the newly created object. To construct a new name for an object, the replay agent modifies the original name by repeatedly appending a suffix until the new name is unique in its directory. To provide the user with a hint about the origin of the new file, the suffix reflects the type of operation being performed. The replay agent keeps track of such renaming, so that later operations on the same object are directed to the right file name. For example, if the replay agent is forced to store changes to `foo` in the new file `foo.ren`, later attribute changes to `foo` are applied to `foo.ren`.

Sometimes it is not possible to replay the operations as they were performed while disconnected. For example, if a file and its parent directory are removed from the file server, modifications to that file by a disconnected client will have no place to be stored. Yet preservation of data is a key goal of our replay algorithm. To handle these problems we provide a centralized location called the ‘‘orphanage’’ to store such files.

5.2.1. `afs_create`

If the parent directory no longer exists, then the replay agent creates the file in the orphanage. If the parent exists and the file already exists, then the replay agent iteratively appends `".creat"` to the name enough times to assure uniqueness, and creates a file with the name that results. If there is no conflict, then the cache manager creates a file in the normal manner.

5.2.2. `afs_write`

Although `afs_write` operations modify the client cache, writes are not propagated to the file server until the file is closed. This means that we can ignore `afs_writes` during replay and be assured that the right thing will happen when the file is closed.

5.2.3. `afs_close`

We can safely ignore any `afs_close` operations performed on files opened only for reading. If the file was open for writing, modifications to the file must be propagated back to the file server. We compare the version number of the cached file to the server’s version number for the file. If they are the same, then the file has not changed since we cached it, so we store our changes.

If the version number of the file has changed, then we do not overwrite the copy on the file server, not even if we are convinced that the local user is the ‘‘last writer.’’ Instead, the replay agent creates a new file with a unique name and copies the data from the cache into this new file and notifies the user.

If the parent directory was removed while we were disconnected, a new version of the file is created in the orphanage, and the cached data is copied into this new file.

5.2.4. `afs_mkdir`

The `afs_mkdir` call is similar to `afs_create`: the replay agent must ensure that no entry in the parent directory has the same name as the directory being created. If there is no conflict, then the `afs_mkdir` can proceed normally.

If there is a conflict, then a unique name is generated and a directory is created with this new name. If the parent directory no longer exists, then the new directory is created in the orphanage.

5.2.5. `afs_remove`

To replay `afs_remove`, the replay agent compares its version number of the file being removed with that on the server. If they are the same, the file is removed. If they differ, the file has been modified. We must not remove it as this would destroy someone else’s fresh data.

To allow files to be removed without having corresponding attribute information in the cache, we invent an invalid version number as the cached version number. In this case, `remove` fails on replay, forcing the user to reissue the `rm` command.

5.2.6. `afs_rmdir`

Because only successful operations are logged, the locally cached copy of the target directory must have been empty at the time the `rmdir` operation was issued. Therefore, the replay agent can simply issue the `afs_rmdir` operation, assured that the operation will succeed or fail on the server appropriately. If the operation fails, the failure status is reported to the user.

5.2.7. `afs_link`

Like `afs_create` and `afs_mkdir`, if the target does not yet exist, then `afs_link` proceeds normally. Otherwise, we generate a unique name and create the link with this new name.

5.2.8. `afs_symlink`

This is the same as `afs_link`, except a symbolic link is being created instead of a hard link. The replay agent needs to ensure that the name is unique, then create the symbolic link.

5.2.9. `afs_fsync`

This operation forces modified data to be stored on the file server. The replay agent must perform the same consistency checks as for `afs_close`.

5.2.10. `afs_setattr`

In replaying `afs_setattr`, the replay agent needs to make sure that no other changes to the attributes are lost. It needs to compare the current attribute information with the cached attribute information. If there are no differences the replay agent assumes there is no conflict and proceeds with the `setattr`.

If there are differences between the current version and the cached version, the `setattr` is aborted and the user is informed of the conflict. It would be possible to merge the two versions of the new attributes if they do not conflict, but this might lead to some unpredictable side effects. Instead, we abort the attempt and alert the user.

5.2.11. `afs_rename`

The `afs_rename` operation deletes the target if it already exists; the cache manager logs whether this was the case when the `afs_rename` operation was initially issued. If the target did not exist, and if a fresh target does not exist on the server when the replay agent runs, then the `afs_rename` operation is issued to the server. If the target does exist on the server, then its name is modified to make it unique, and the `afs_rename` operation is run with the new target name.

As with `afs_remove`, the disconnected cache manager allows a `rename` if a destination entry exists but no information for this entry is cached. In this case, the destination version number is marked as invalid so replay will fail to destroy this file.

6. Data Persistence

One problem encountered in our AFS modifications is the amount of state that the cache manager builds while talking to the servers. The cache manager stores each object in two components: the `vcache`, which holds status and callback information associated with the object; and the `dcache`, which holds the information needed to translate AFS' internal name for a file into a name in the local cache.[3]

AFS needs a `dcache` for every file in the cache. There are too many to hold in kernel memory, so the cache manager stores most `dcaches` on disk and moves them into a memory cache as needed. `vcaches` are not as critical, so the cache manager keeps a large cache of them in kernel memory. AFS was designed to run on high-speed networks, so the expense of fetching `vcaches` from the file server has traditionally been negligible. But in a disconnected environment, it is not possible for the cache manager to ask the file server for the current `vcaches`, so we store `vcaches` on disk just as we do `dcaches`, and move them into memory when they are referenced. This also allows `vcache` information to survive reboots.

In addition to the `vcache` and `dcache` entries, the cache manager keeps other important information in volatile memory. The AFS namespace is constructed from subtree components called *volumes* [16], which are mounted together to form AFS' hierarchical name space. To cross a

3 This internal name, called a FID, is a data structure consisting of a cell identifier, a volume name, a vnode number, and a version number, *e.g.*, `<8DD3A818, 200024A1, 1C8A, 12>`. The local file name looks like `/usr/vice/cache/V1001`.

mount point, the client must obtain a mapping from a volume name to a directory identifier and a server name. Ordinarily, this mapping is obtained from a volume location database server. Once a mount point has been crossed, the client caches the mapping to speed subsequent lookups. However, this information is kept in volatile memory, so it does not survive a reboot.

The cache manager needs these volume mappings and other memory based data structures to access cached files. For mobile clients to continue working across reboots, we had to make these data structures persistent, again using disk files as backing stores for these data structures. These data structures are loaded into memory from the disk files as part of the AFS start up procedure.

7. Current Status

We are running disconnected AFS on our laptop machines and use it on a daily basis. We have most of the bugs worked out and are experimenting with disconnected operation to help determine what features need to be added. Disconnected AFS was used in writing portions of this paper.

8. FUTURE WORK

There are several areas that we plan to address in the near term. One issue is callback management. We would like to add support to the cache manager so that when it is connected it tries to keep the most up-to-date version of the files in our cache. This will help reduce potential replay conflicts, because we won't start by using stale data.

Another issue that needs to be addressed is the cache replacement policy. AFS currently uses a least-recently-used (LRU) policy to determine which items in the cache should be replaced. We will use our nomadic experiences to determine if this works well enough for our needs. We may need to be able to ‘‘pin’’ certain files in the cache, *e.g.*, system binaries used to set up a network.

We also plan to provide tools to help users resolve conflicts. It is sometimes possible to determine an effective heuristic behavior for a specific application. For example, we make heavy use of the `MH` mailer, which stores each mail message in a sequentially numbered file. If messages are inserted simultaneously by a connected client and a disconnected client, we might end up with a message file called `1234.creat`. Yet we know how to fix matters: rename the file to an unused number. We plan to write scripts to address this and other application-specific scenarios that we encounter.

Another area that needs work is management of the log file. When running disconnected, many operations are overruled by later operations. For instance, if we create a file while disconnected and remove it before we reconnect, there is no point in storing the file during the replay. We plan to write a program that examines the log and removes this and other extraneous operations. This will have multiple benefits. First, it will reduce the disk space needed by the log. Second, it can reduce the time necessary for replay, and might make replay more palatable over slow or expensive data links.

Another problem we face is how to react to the use of stale data. Ideally, we would like to be able to identify all the files and processes that depended on this stale data, so that the user may be able take some corrective action, such as rerunning the processes on fresh files. To provide this functionality, we would need support for transactions, which could provide us with the information necessary to detect these read-write conflicts.

9. Conclusions

Disconnected operation vastly increases the benefits of nomadic computing. Currently, to use a LITTLE WORK machine, a user needs only to use the laptop on a network for a LITTLE WHILE and the machine is ready to roll. If she performs work similar to what she intends to do on the road, the cache will contain all the files necessary to support her needs. At this point she may tell AFS to run disconnected.

While traveling, the user can use the laptop as if she were sitting at her desk. Files and directories are all in their usual places. When arriving back home or in a hotel room, the user can establish a network connection and tell the cache manager to reconnect, whereupon all changes are propagated back to stable storage. As a result, the nomadic computing environment closely mimics the conventional one, vastly extending the range and scope of mobile computing.

Acknowledgements

We thank Jim Rees, Mike Stolarchuk, Mary Jane Northrup, and M. Satyanarayanan and for their insight and assistance. This work was partially supported by IBM and Telebit.

References

1. P. Honeyman, L. Huston, J. Rees, and D. Bachmann, ‘‘The LITTLE WORK Project,’’ *Proceedings of the Third IEEE Workshop on Workstation Operating Systems*, Key Biscayne, FL (April 1992).
2. Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, ‘‘Mach: A New Kernel Foundation for UNIX Development,’’ *USENIX Conference Proceedings*, Atlanta, GA (Summer 1986).
3. R.W. Scheifler and J. Gettys, ‘‘The X Window System,’’ *ACM Transactions on Graphics* **5**(2) (April, 1987).
4. J.L. Romkey, ‘‘Nonstandard for transmission of IP datagrams over serial lines: SLIP,’’ RFC 1055, Network Information Center, SRI International, Menlo Park, CA (June 1988).
5. D.L. Mills, ‘‘Network Time Protocol (Version 3): Specification, Implementation, and Analysis,’’ RFC 1305, Network Information Center, SRI International, Menlo Park, CA (March 1992).
6. J.G. Steiner, B.C. Neuman, and J.I. Schiller, ‘‘Kerberos: An Authentication Service for Open Network Systems,’’ *USENIX Conference Proceedings*, Dallas, Texas (February, 1988).
7. John H. Howard, ‘‘An Overview of the Andrew File System,’’ *USENIX Conference Proceedings*, Dallas, TX (Winter 1988).
8. M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere, ‘‘Coda: A Highly Available File System for a Distributed Workstation Environment,’’ *IEEE Transactions on Computers* (April 1990).
9. J.J. Kistler and M. Satyanarayanan, ‘‘Disconnected Operation in the Coda File System,’’ *ACM Transactions of Computer Systems* **10**(1) (February 1992).
10. P. Kumar and M. Satyanarayanan, ‘‘Log-Based Directory Resolution in the Coda File System,’’ *Second International Conference on Parallel and Distributed Information Systems*, San Diego, CA (January 1993).
11. J.S. Heidemann, T.W. Page, R.G. Guy, and G.J. Popek, ‘‘Primarily Disconnected Operation: Experiences with Ficus,’’ *Proceedings of the Second Workshop on the Management of Replicated Data* (November 1992).
12. 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,’’ *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (December 1985).
13. Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, ‘‘Measurements of a Distributed File System,’’ *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA (October 1991).
14. A.M. Khandker, ‘‘Mobile Computing: Running AFS over Dial-up Connections,’’ CITI Tech. Report, University of Michigan (In preparation).
15. S.R. Kleiman, ‘‘Vnodes: An Architecture for Multiple File System Types in Sun UNIX,’’ *USENIX Conference Proceedings*, Atlanta, GA (Summer 1986).
16. R.N. Sidebotham, ‘‘Volumes: The Andrew File System Data Structuring Primitive,’’ *European Unix User Group Conference Proceedings* (August 1986).

Availability

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