A Toolkit Approach to Partially Connected Operation

Dan Duchamp
Columbia University and AT&T Labs

A Toolkit Approach to Partially Connected Operation

Dan Duchamp

Columbia University
and
AT&T Labs

Abstract

Partially connected operation is the circumstance in which the communication link between two computers is intermittent, either by choice or because or failure. This paper describes the design and performance of a user level toolkit that is suited to accessing the home directory of a partially connected user in a bandwidth-efficient manner.

Compared to custom file systems for partially connected operation, the toolkit is easier to deploy and provides extra degrees of flexibility because it runs at user level and uses the local file system for its cache. The toolkit makes it possible for unaltered clients to access NFS file systems exported by unaltered servers.

The maintenance of consistency between client and server is automatic provided that certain assumptions are upheld, the primary one being that sharing is limited in such a way that there is a "single locus of update." That is, for extended periods, updates are applied either to the client's cache or directly to the server, but not to both simultaneously. This pattern of use is typical of a user's home directory.

Performance is the disadvantage of user level operation. The client's cache is managed by a "caching tool" that services every file system operation, and this redirection increases latency substantially.

1 Introduction

This paper explains a set of simple user level tools that is suited to accessing one's home directory over an intermittent connection. The toolkit implements "partially connected operation" for standard file system protocols (e.g., NFS version 2) running on unmodified UNIX hosts. The work was motivated by the following personal experience, which provides an example of partially connected operation.

My home computer has a leased line connection to an Internet service provider (ISP); my company uses a different ISP. In order to avoid being a system administrator, I initially used my home computer as an X terminal, `telnet`-ing to the machines at my office, Unfortunately, network response is highly variable—waiting seconds or minutes for keystroke echo is not unusual, outages are frequent (e.g., several times per evening) and sometimes long (e.g., an entire holiday weekend)—enough so that interactive operation over the Internet proved impractical.[1]

I considered two alternatives. One was to continue using the home computer as a terminal, dialing up to the office. The other was to make the home computer as self-contained as possible. My home is located in a different area code from my office, so dialup accrues long distance charges both per-call and per-minute; therefore, I chose the second alternative.

I implemented a "toolkit" of user-level programs that together form what might be viewed as a poor man's version of a distributed file system. The system minimizes the connect time between the client at home and the server at work, hence "partially connected operation." Compared to the pioneering work on file systems supporting disconnected and/or partially connected operation

[1] The primary cause of the problem is that home and office have different ISPs. The two ISPs connect at the "MAE West" router on the west coast, so packets between New York City and northern New Jersey—four miles as the crow flies—travel across country and back again, typically going through 20 hops.

(e.g., [9, 14, 6, 8]), what I have implemented is unambitious and not technically groundbreaking. However, over time it became clear that operating at user level—where tools such as file system utilities, scripting languages, etc. are available—brings some profound advantages. My toolkit approach stands in contrast to the approach based on custom kernels and file system protocols. My claim is that a simple user-level toolkit can handle the most common workload (i.e., files drawn from a personal home directory, with updates coming from only one side of a connection at any time) with considerably less mechanism, more flexibility, and much greater ease of deployment.

In my approach, the home computer is a file system client that is disconnected from the servers at work except when a phone call is made between them. The derived technical problem is how to keep the home and office copies consistent while minimizing phone charges. The toolkit is used as follows.

1. The office file server (master) exports my home directory to the home computer (slave) as a separate NFS file system.[2]

2. The slave caches a subset of my home directory and also has a normal collection of slowly changing system files. Cached files are kept not in a hidden area, but in the local file system, in the same hierarchy, with the same names, as at the master. When I am at home applications execute on the slave, accessing files using the same names as if the application were running at the office, but the files come from the cache.

3. The slave fetches files and directories on demand, using the NFS protocol (version 2); however, it does not mount the master's exported file system in the usual way. Instead, the master file system is mounted so that a local process is its NFS server. Whenever the slave accesses a file with a name that is in my home directory, the process —called the "caching tool"— receives control. When possible, it redirects the access to the local cached file with the same name. If the relevant file is not yet cached, the process fetches it and copies it to the local cache.

   The caching tool is a modified version of Amd [17] (version "upl 102"), a user level automounter. Partially connected operation is implemented as a new file system type (named "cfs"). A UNIX client need only upgrade to the new version of Amd to get partially connected access to home directories as well as all other services currently supported by Amd.

4. When I move between office and home, the file hierarchy at the new location will be out of date with respect to the old location if I made changes on the old system. To determine the changes to propagate from the old system to the new, I run on each system a "checksum tool" that computes cryptographic checksums for the file hierarchy on that system.

   The checksum of a directory is computed over the names and checksums of the directory entries. If two corresponding directories have the same checksum, the entire hierarchies rooted at those directories are identical.

5. The checksum tool produces a report that is fed into a "comparison tool." This tool uses the report plus knowledge of which site has the most recent changes to decide which files and directories to copy to or delete from the site that is out of date.

   The comparison tool achieves massive pruning from the fact that equality of two directory checksums implies equality of the entire hierarchies rooted at those directories.

This approach does not constitute a traditional distributed file system. Rather, it is a toolkit appropriate only in the case where there is an easily identified "single locus of update"—i.e., for extended periods, updates are applied either to the home file system or to the office server, but not to both. The checksum and comparison tools are invoked whenever the locus of updates changes, and together they reestablish consistency. Neither file system should accept updates while the checksum and comparison tools are running. Since home systems and portable computers are intimately personal, the limitation might not be as severe as it may seem at first. Indeed, experience with the Ficus file system in a similar setting of a home workstation suggests that there is only a single locus of updates—that "the user acts as a write token" [4, 18].

While the single locus of update restriction significantly reduces the technical problem addressed, the toolkit nevertheless is interesting for a few practical reasons. First, the one case it handles (a single easily identified locus of updates)

[2] At my site, this required some administrative changes. Ordinarily, several home directories are bundled into a single file system.

is a common, and perhaps the dominant, read-write file system workload. Further, no kernel changes are required at either client or server, and only a few configuration changes are needed. The major ones are, first, to have the server export the home directory as a separate file system, and, second, to make the caching tool service the mount point at the client. All the tools run at user level; only the caching tool must run with root privilege. A consequence of the easy deployment is that for the first time standard file systems (e.g., NFS) are available to ordinary, partially connected UNIX clients. Finally, this work is highly portable. The checksum and comparison tools are "ordinary UNIX programs" that need no special privilege. The most complicated component, the caching tool, is a version of Amd, which is one of the most widely ported UNIX programs in existence. The caching tool inherits this high degree of portability. Perhaps ironically, the tool that is by far the hardest to port is the dialup software.

The next section explains the operation of the tools. The section after that presents performance analysis. Section 4 discusses related work.

2 Tools

The next few sections explain the design and operation of the tools for checksum computation, file hierarchy comparison, caching, and dialup.

2.1 Checksum Tool

The checksum tool writes a file — named "`.contents`" — in each directory of a hierarchy. Such a file contains the type of checksum algorithm used and the names and checksums of each file, directory, or symlink in the directory as of the time the tool was invoked. The name/checksum entries are sorted by name. Other objects that can populate directories (like devices and sockets) are ignored. Any pre-existing `.contents` file is also ignored.

The checksum of a symbolic link is computed by *not* following the link but rather by checksumming the contents of the link. Not following symlinks converts a file system hierarchy from a general graph into a DAG. The DAG is then treated as a tree; i.e., a file with multiple links appears several times in the tree, once in each directory. The checksum of a directory is the checksum of its `.contents` file; therefore, checksumming is a bottom-up operation.

As an example, suppose a directory contains two files `w`, `x`, a symlink `y`, and a directory `z`, with checksum values 1, 2, 3, and 4, respectively. The following text would be checksummed and the resulting value considered to be the checksum of the directory:

```
algo MD5
w 1
x 2
y@ 3
z/ 4
```

In this case, 3 is the checksum value of the symlink stored in `y`. The "`@`" and "`/`" symbols denote symlinks and directories, respectively. The comparison tool uses this type information, as it may need to treat symlinks and directories differently.

The available checksum algorithms are MD4 [19], MD5 [20], and UNIX "sum." The checksum algorithm is a modular component of the tool, and it is trivial to add new choices.

An entry that is unreadable is considered not to exist and is not listed in `.contents`. If a directory is unwritable, then its checksum field is left blank in the `.contents` file in the directory above.

If the two file hierarchies to be compared are on hosts linked by a distributed file system (such as NFS), then the comparison tool can simply read the `.contents` files in corresponding directories of the two hierarchies and perform its comparisons based on what it reads. However, in some environments it may be impossible or impractical for the two hosts to be linked by a distributed file system protocol; instead, some other protocol (such as FTP) is used for short-lived and/or limited data shipping. To support such environments, the checksum tool optionally can also dump all directory checksums — and *only directory* checksums — into a "dumpfile." This file, which is expected to be relatively small compared to the entire hierarchy it summarizes, can then be sent from one host to the other, and the receiving host invokes the comparison tool, which operates as described below.

2.2 Comparison Tool

The comparison tool has a few different modes of operation. The command line specifies the mode as well as which hierarchy has been the most recent locus of updates.

The checksum tool may have written `.contents` files or a dumpfile, and it may or may not be possible for the comparison tool to access both hierarchies via normal UNIX calls such as `open()` and `fread()` (i.e., both hierarchies are accessible as file systems).

Regardless of which case is exercised, comparison of checksums proceeds top-down, with every match pruning the comparison. In the most extreme case, the checksums of the root directories of the two hierarchies match, and a single comparison concludes that the entire hierarchies are identical. More typically, one or both sides will have entries that the other side does not have.[3] In these cases, and in cases where both sides have an entry but with different checksums, the comparison tool emits UNIX commands that make updates, creates, and deletes in order to bring the outdated hierarchy up to date.

Large files with holes are handled specially, as explained in Section 2.3.3: a special copy program is used to transfer only sections of a large file.

2.2.1 Precious Files

The term "precious file" is taken from `rdist` [1]. At my site there is one significant exception to the single locus of updates behavior. We use `hlfsd` [26] to deliver Email into a user's home directory.[4] Regardless of whether the user is at home or work, Email goes into a file on the server at work.

Special purpose protocols like POP3 [21] or IMAP4 [2] exist to remotely manipulate mailboxes, so the comparison tool does not change the mailbox (and any other such precious files), leaving them for the special protocols.

2.2.2 Fault Tolerance

Assuming that the single-locus restriction is maintained and that checksumming and comparison is done between locus changes, a failure during checksumming or comparison impacts performance but not correctness.

There is no state kept outside the client and server file systems. The checksums are derived information, and the copying done by the comparison tool is idempotent assuming that non-precious files are not updated while the comparison tool runs. Therefore, if there is a failure during the checksum/comparison phase, restarting from the beginning will establish a correct state.

The two real dangers are violating the single-locus-of-update restriction and making updates during the checksum/comparison phase. Section 2.4 discusses the steps that can be taken against these dangers.

2.3 Caching Tool

The caching tool is an altered version of the latest release of the Amd automounter, a program written by J-S. Pendry [17].[5] Many of the alterations were taken from the Autocacher, which was written by R. Minnich [12]. The Autocacher is itself an altered version of an old release of Amd. I ported the Autocacher features into the latest version of Amd, then added my features.

This section explains the new features, assuming that the reader understands how Amd and Autocacher operate. Readers lacking this background can find it in Appendixes A and B.

For file systems of type "cfs" (a name taken from Autocacher, denoting "caching file system"), the caching tool implements all NFS operations. The primary data structures and the implementations of `read`, `readdir`, `readlink`, `lookup`, and `getattr` are largely those ported from the Autocacher. The new operations are "mutating" operations that make updates to the local cache; they can be classified into categories: delete (`remove`, `rmdir`, `rename`), create (`mkdir`, `create`, `rename`), and update (`setattr` and `write`). Note that the rename operation falls in two categories.

It is undesirable to have every file system operation leave the kernel to go through the caching tool; Section 3 shows the negative impact on performance. However, redirection of all operations through the caching tool is necessary, for these reasons:

1. The caching tool needs to know about deletes.
2. Because of the way deletes are represented in the file system, the caching tool must also know about creates. If a create operation recreates a deleted name, the delete indicator must be deleted. Section 2.3.2 elaborates.
3. The caching tool must be in control of read operations in order to intelligently transfer large files (see Section 2.3.3).

[3] The `.contents` files are sorted in order to speed the task of detecting entries that are in one `.contents` file but not the other.

[4] `hlfsd` controls the name space of the system mail spool area — e.g., `/usr/spool/mail` — and converts the name of each user's mailbox into a symlink into the user's home directory. For example, `/usr/spool/mail/user` becomes a symlink to `$HOME/.mailspool/user`.

[5] J-S. Pendry wrote the original Amd. Erez Zadok merged patches submitted by many persons and into the latest version, named "upl 102." [25]

These reasons justify the caching tool handling some, but not all, NFS operations. It handles all operations because there is no way — without special kernel support such as stackable vnodes [5] — to selectively redirect a subset of operations through the caching tool while letting others go directly to the native file system. The caching tool must return a file handle on every lookup, and this handle is used for all operations thereafter.

There are four interesting aspects to the implementation: copy policy, deletes, large files, and cache management.

2.3.1 Copy Policy

The rule, taken from Autocacher, is that files are copied from the master lazily but directories are copied eagerly. This means that a lookup on an uncached directory results in the directory being copied and the returned handle pointing to the local copy, whereas a lookup on an uncached file does not result in a copy. The returned file handle is one fabricated by the caching tool and used for emulation. Files are cached lazily because it is common for a file to be looked up but not accessed. A lazy caching policy avoids transferring every file that is shown by an `ls` command, for instance.

The decision to treat files and directories differently is founded on two assumptions. First, that the number of directories and the number of bytes occupied by directories are both small compared to the same numbers for files. Second, that the directory structure of a user's home area changes slowly. The first assumption implies that it does not cost much to copy uncached directories to the client. The second assumption implies that, once cached, a directory is likely to stay at the client.

2.3.2 Tracking Deletes

The delete operations require some care. The reason is that files and directories are demand-fetched from the master, meaning that the slave has only a subset of the master's contents. So when the comparison tool finds that a file or directory exists at the master but not at the slave, there is potential ambiguity: has it been copied to the slave and deleted there, or never copied to the slave?

I considered two approaches to resolving the ambiguity. The first is to perform every delete at the master as well in the slave's cache. This removes the ambiguity: a file or directory present at the master and absent at the slave has not been copied to the slave. However, this approach increases the number of interactions between slave and master, which is potentially costly if the network connecting the two is pay-per-use.

The second approach, which I used, is to maintain a delete log. With this approach, the ambiguity is removed in the opposite way: if a file or directory is present at the master and absent at the slave, it has been deleted at the slave if and only if the slave has a record of the deletion.

I maintain what might be considered a distributed log: deleted files and directories are renamed in the cache. The renamed files and directories are called "deletion markers."[6] As shown in Figure 1, deleted files are truncated and renamed with a special prefix. A deleted directory is similarly renamed provided that it contains only deletion markers. Before being renamed, the deletion markers in the directory are all deleted; Figure 2 illustrates.

The create operations make new directories and files as needed in the local disk cache, but they must be aware of deletion markers. When `create` creates a file, it checks for the associated deletion marker and deletes it if found, then creates the file. Checksumming will detect any conflict between the new file in the cache and the old file back at the master. Other interactions between create and delete operations are more subtle.

One subtle interaction is that deletion markers must be removed from a deleted directory so that later operations will proceed correctly. Suppose otherwise: that a deleted directory were simply renamed as a marker, and the markers within it were preserved. If the same name is later `mkdir`-ed and `rmdir`-ed again then the `.deleted.DIR.` name will exist and will not be empty at the time of the second `rmdir`. In this case the rename will fail because the marker directory will not be empty.

A second subtle interaction between creation and deletion is that in contrast to `create`, `mkdir` *does not* delete an existing deletion marker that corresponds to the name it is creating. Once a directory has been deleted at the slave, its deletion marker remains, even if the directory is re-created. The deletion marker will remain until it is deleted by the comparison tool. If a directory's deletion marker were deleted when it is re-created, then the sequence `rmdir foo ; mkdir foo` would result in a state indistinguishable from not having cached the files in `foo`, and the comparison tool would fail to delete the files on the master. Retaining both the new directory and the deletion marker indicates that it was deleted and recreated. Delet-

[6] A similar mechanism is the "whiteout" used for 4.4BSD union mounts [16].

```
Before deletion:
    -rw-r--r--  1 djd  faculty  7183 Jun 14 22:26 foo
After deletion, foo is gone and a marker remains:
    -rw-r--r--  1 djd  faculty     0 Jun 14 22:26 .DELETED.foo
```

Figure 1: File Deletion Using Markers

```
If before deletion:
    drwxr-xr-x  2 djd  faculty   512 Aug 17  1995 src
contains:
    -rw-r--r--  1 djd  faculty  7183 Jun 14 22:26 foo
    -rw-r--r--  1 djd  faculty     0 Jun 14 22:26 .DELETED.bar
Then rmdir will return an error because foo still exists.
However, if before deletion the contents are:
    -rw-r--r--  1 djd  faculty     0 Jun 14 22:26 .DELETED.foo
    -rw-r--r--  1 djd  faculty     0 Jun 14 22:26 .DELETED.bar
Then rmdir will succeed and result in the empty directory:
    drwxr-xr-x  2 djd  faculty   512 Jun 14 22:27 .DELETED.DIR.src
```

Figure 2: Directory Deletion Using Markers

ing the re-created directory requires removing the directory while leaving the deletion marker.

2.3.3 Large Files

As described to this point, emulation is accomplished by copying the whole file from master to slave, then having the slave's caching tool operate on the local copy. If the file is large, and the communication link is slow and/or costly, copying the whole file can be expensive and cause substantial delay. For example, copying a 1MB file over a telephone line with effective throughput of 2.7KB/sec—75% of 28.8Kb/sec, an optimistic scenario—takes well over 6 minutes. Accordingly, the toolkit handles large files specially, transferring, checksumming, and comparing smaller units.

When the caching tool "copies" a large file in fact what it does is copy the file into a special marker file in the same directory and make the file's name be a symlink pointing to the marker.[7] The caching tool notes that a file is a large one and handles reads and writes specially. Read operations transfer only the requested byte range, assuming it is not already cached. Write operations also copy the requested byte range, if necessary, then apply the update.

The physical file in the cache is kept in a special format. The caching tool writes a header region at the beginning of the file that describes what ranges of the master file have been copied into the cache, and at what offset each range can be found in the cached file. (This special handling of large files is an example of the flexibility that can be achieved with user-level tools.)

When the checksum tool runs, it will always compute different checksums for the large file's name at client and server, because the name will be a file at the server but a symlink at the client. When the comparison tools detects a checksum mismatch, and detects that the client name is a symlink whereas the server name is a file, it then inspects the symlink value to determine if it is a marker for a large file. If it is, then a special copy utility reads the header region of the cache file and copies to the master file only the byte ranges that the cache file holds.

The definition of a "large" file is a parameter that can be set either on Amd's command line or via `amq`, the configuration utility for Amd. The

[7] The name of the marker file is the file's name preceded by "`.large-file.`".

parameter is a number of seconds. The caching tool measures the network throughput achieved by recent copy operations. A file is "large" if copying it would take more than the specified time.[8] By setting the time parameter to zero, the user can treat all files as large files. The reason not to do is that the checksum/compare phase will take substantially longer than usual if every file is a checksum mismatch.

2.3.4 Cache Management

Storing the cache and the "log" in the local file system contrasts with the approach taken by specialized in-kernel file systems that implement the log and cache as separate data structures, both outside the file system name space.

With the cache in the local file system, ordinary utilities and scripts can be used as cache management tools. This toolset can be superior in flexibility, scope, and ease of use compared to the mechanisms that file system implementors can include into the client side of an in-kernel file system. Therefore, it may be easier to implement complex, user-specific and/or time-varying criteria for removing files or handling special cases. For example, cache files can be deleted based on any criteria any time the cache is consistent with the master. (I put cache cleanup in the logout script before the checksum phase.)

Coda and AFS both maintain a log of all mutating operations. The most important purpose for a log is to record deletes, but log information is also exploited for attempted automatic resolution of conflicts. In both systems, the log is a single data structure, and managing it is a source of complexity. If the log fills, processing stops; however, for typical workloads, information early in the log is superseded by later log entries. This motivates techniques such as log compression [7] and "trickle discharging" [14]. In my approach, the "log" and the cache share the same storage, so neither has a size limit that might not be ideally matched to the other's size.

2.4 Change of Locus

Change of locus raises three issues: how to detect it, exactly when to execute checksum operations, and how to detect that the single-locus-of-updates assumption has been violated.

[8]The first file, accessed before there is any network throughput information, is deemed large if it exceeds 50KB.

Change of locus is detected by running a script. The script runs when the user arrives at the new site or departs the old site. It can be executed by the user or can be invoked automatically as part of login/logout or as part of a terminal locking program, etc. The tradeoff between running the script at the new or old site is performance versus safety. If the script is written assuming that the user is leaving his old site, the checksum operations can be overlapped with the user's movement, but there must be some means to indicate that the script is finished (because the user must refrain from making updates at the new site until the compare/copy operation is finished). I have not implemented such a scheme, instead doing one checksum operation at logout and the other at login.

The compare phase takes time proportional to the number of updates at the most recent locus. The checksum phase takes time proportional to the size of the file hierarchy, and the constant factor is much larger than for comparison. Therefore, checksumming the master is by far the longest operation. Accordingly, I use login/logout as the indication of locus change, checksumming the master at logout and running the slave checksum and comparison at login.

Invocation of checksumming at the other site is accomplished with `rsh`.

Violation of the single-locus assumption can be detected simply. In particular, when checksums are computed at the new locus, existing `.contents` files should not be overwritten. Instead, they should be moved, say, to "`.previous-contents`." Then the comparison tool not only compares corresponding `.contents` files at the two locations, but also compares `.previous-contents` and `.contents` at the new locus. If they disagree, the single-locus assumption was violated; i.e., updates were made at both the new and old loci.

Note that consistency re-establishment need not occur only at change of locus. That is, a home user can invoke the tools in the middle of his session *as if* a change of locus were underway, provided that he refrains from updates for the necessary period of time.

2.5 Communication Software

Although dialup is a mundane function, the dialup package is the greatest obstacle to the toolkit's portability and deployment.

Dialup is a portability problem because what is needed is *on demand* dialup. That is, a telephone connection must be brought up automatically whenever there is outgoing traffic; dialing upon login, say, and holding the connection defeats the purpose of the toolkit. Many UNIX TCP/IP stacks will simply return an error if there is not already an active link device ready to accept outgoing traffic. Providing a hook to allow a user level script to set up a link on demand is an operating system issue, thereby making it hard to formulate a solution that is as portable across UNIXes like Amd. My home client runs Linux. Linux has the `diald` on-demand dialup package [24] that provides the right function. This is what I use.

Dialup can be a deployment problem because machines on both sides of the communication link need to dial each other. Most installations have firewalls, many have restrictions on acceptable sources of dial-in and targets of dial-out, and many elect not to support SLIP or PPP access. These restrictions can be overcome, but may increase the amount of administrative changes that must precede using the toolkit.

An issue for the script that sets up and tears down the telephone connection is when to tear down. In a slightly different context this has been called the "Holding Time Problem" [22]. The issue is that maintaining an open connection costs money, but so does restarting a closed connection, so shutting down immediately is not necessarily the best policy. Tariffs are known and, if future traffic patterns were known, a simple calculation would indicate how long an open connection should be maintained. However, there is no realistic way to predict future traffic, especially from a single host. Accordingly, I have changed `diald` to include the heuristic that the telephone connection is shut down after an idle period whose connect charge equals the charge to make a new phone call. This policy limits the overall cost of a phone call to twice the charge to make the call (the real charge for making the call plus the charge for the idle time that causes the shutdown) plus the charge for the time used. Here time "used" means real use plus all idle periods shorter than necessary for a shutdown.

It is desirable to replace the dialup package with a more powerful "communication tool" that could direct traffic to my Internet connection opportunistically, falling back to the phone line only when the Internet connection becomes unacceptable. Such a tool is even more operating system specific than a dialup package. Work is underway on a prototype.

TYPE OF LOOKUP	LATENCY
NFS, uncached	6.4 msec
NFS, cached	2.7
Caching tool, uncached	49.3
Caching tool, cached	49.2
Cachefs, uncached	10.4
Cachefs, cached	3.4

Table 1: Cost of Redirection, Lookup

3 Evaluation

The next few sections report performance measurements. Section 3.5 summarizes the administrative changes that might be needed to deploy and use the toolkit on UNIX.

3.1 Basic Cost of Redirection

The first measurement determined the basic cost of redirection. I did a number of `getattr` operations, measured the overall time to do them as well as the time spent in the caching tool, then subtracted. The result is the amount of time it takes to go into the kernel, have the kernel redirect the NFS operation to the caching tool, which acts as the NFS server, then return the result.

On a Sun SparcStation-2 running Solaris 2.4, the result is 38.1ms with a standard deviation of 3.2ms.

3.2 Added Cost of Emulation and Caching

The second experiment determined the time needed to do certain common NFS operations such as `lookup` and `read`. The caching tool was timed as was regular NFSv2 and Sun's "cachefs" file system, an in-kernel file system that caches on disk.

In the lookup test (Table 1) the parent directory had already been looked up. The reported times are end-to-end measured from a user level program. An "uncached" lookup goes to a remote file server. In each test, the standard deviation was very small.

In the read test (Table 2) the file had already been opened and one 4KB block was read. The reported times are end-to-end measured from a user level program. An "uncached" read goes to

TYPE OF READ	LATENCY
NFS, uncached	6.7 msec
NFS, cached	5.4
Caching tool, uncached	175.2
Caching tool, cached	76.2
Cachefs, uncached	26.7
Cachefs, cached	21.1

Table 2: Cost of Redirection, Read

a remote file server. A cached read is drawn from memory in the case of NFS, and from disk in the case of the caching tool and Sun's Cachefs. In each test, the standard deviation was very small.

The very slow times for the caching tool show the effect of copying data between a process and the kernel. For the uncached read via the caching tool, the data is copied from the remote server to the caching tool, written from the caching tool into a local file, then copied back into the kernel as the result to return to the requester.

3.3 Checksum Speed

I ran the checksum test with my home directory as input. My home is on a faster platform (SparcCenter-1000 running Solaris 2.5). I measured 4.6ms per file with MD4 and 6.2ms per file with UNIX sum. For my home directory of 7502 files comprising 899MB, it takes MD4 close to 35 seconds to compute checksums.

3.4 Accuracy of Single-Locus-of-Update Assumption

The effectiveness of the toolkit rests, in large part, upon the assumption that home directories typically have a single locus of updates except for a few easily identified exceptions such as Email delivery.

Numerous studies have shown that there is very little write sharing in file systems; for example, [9] examines the number of times consecutive updates are made by different users. The data is suggestive, but "different users" is not the issue. It could be that the same user is making consecutive updates, but from different sites. If one of these sites becomes disconnected yet still makes updates, there would be conflict even though the same user made all the updates.

The toolkit requires the stronger condition that while updates are made at home they are not made at the office server, and vice versa, for the user's entire home directory. I am aware of no study that measures usage patterns with sufficient precision to decide how dangerous the single-locus-of-update assumption is. One complicating factor is that it is acceptable for updates to come from any number of office-side client machines during the time when the locus of updates is at the office.

To shed some light on the matter, I changed the office-side mount daemon so that it timestamped its records of remote mount/unmount operations. Using the time information one can tell at all times how many clients had a file system mounted. I have seen no unexplained mounts during the time when home is the locus of updates. However, as discussed earlier, it is possible to detect when a file had been changed at both sites by performing one extra checksum operation and one extra compare.

3.5 Summary of Administrative Actions

The following system administration steps must be taken to setup and use the toolkit for partially connected access to one's home directory. Depending on an installation's current policies, these steps may or may not represent changes.

1. At the server: make the user's home directory a separate file system, and export this file system so that the user's home machine can mount it.

2. At the client: install the caching tool as the automounter; the caching tool can replace Amd version upl 102.

3. At the client: add the "cfs" entry to the Amd maps, and allocate an area of the local file system to serve as cache and log.

4. At the client: install a dialup-on-demand package.

5. At both sites: allow IP dial-in and/or dial-out to the other site. Allow execution of rsh commands sent from one site to the other.

6. The user: write scripts to invoke checksum, comparison, and cache cleanup. Execute these scripts from login/logout or other scripts that execute manually or automatically when the locus of updates switches.

4 Comparison to Related Work

There are two principal categories of related work. One category contains the many commercial products for "file synchronization." Well known products include Symtanec's *PC Anywhere*, Traveling Software's *Laplink*, and Microsoft's *Traveler's Briefcase*. The other category is the published work on "normal" distributed file systems that have been extended to accommodate users who suffer various levels of disconnection. In this category are Coda [9, 10, 11, 13, 14, 15, 23], Ficus [3, 4, 18], and the Michigan alterations to AFS [6, 7, 8].

My work sits in between. The main difference with the commercial tools is that operation is more automatic, no special software is required on the master server, or both. Also, I know of no tools, commercial or otherwise, that use hierarchical checksums. The main difference with the research file systems—besides scope, a special purpose toolkit versus a full fledged distributed file system—is point of view. Perhaps because of the history of the general purpose file system projects, the point developed in many of the papers cited above is that general purpose techniques can be extended and adapted to roughly accommodate the needs of (completely, partially, and mostly) disconnected users. This paper raises and partly answers an opposing question: can a large fraction of the usefulness of distributed file systems for partially disconnected users be delivered through a portable toolkit that has only a small fraction of the complexity? The papers cited above explore whether extension of existing distributed file system techniques is *sufficient* to accommodate disconnected users; this paper raises the issue of whether such techniques are *necessary*.[9]

The main issue is the workload. If the workload upholds the single-locus-of-update assumption, then the consistency problem becomes trivial and the need to build support for consistency protocols, logging, and conflict resolution into the basic file system becomes questionable.

It is widely accepted that most areas of a file namespace can be accurately characterized as either *personal* read-write areas or read-only shared areas. Data gathered from the file system projects suggests that personal areas typically do have only a single locus of updates, and that conflicts arising between replicas can be handled simply, as the comparison tool does. For example, [4] states:

```
Another factor contributing to the
rarity of conflicts is the effect
of a human write token ...
Because updates to personal data
come primarily from a single user,
that person serves effectively as
a ``write token'' for those files.
By arranging a pattern of
reconciliation corresponding to the
presence of the user, the most
recent data is almost always
present when updates are made.
```

The "arranging" referred to is the tweaking of system parameters that govern when the reconciliation function built into the file system runs. One might ask whether—given a fast user level reconciliation procedure—reconciliation should be done that way: by the file system, with involvement of the system administrator. Further, [15] states:

```
In the vast majority of cases,
resolution merely involves
overwriting a stale replica
with the most current one.
```

A third category of relevant work is techniques to automatically resolve conflicts [10, 18, 11]. This work is valuable in its original context and is complementary to my work: a slight generalization of the comparison tool would have it invoke such tools whenever a conflict is found. Operating at the user level allows such tools to become quite elaborate, if necessary.

A final point regarding similar tools is that the function provided by the toolkit cannot be duplicated simply by `rdist`-ing from one site to another. The difference is that the caching tool copies on demand from the master and tracks deletes. Copying on demand is more bandwidth-efficient, and the tracking of deletes allows the comparison tool to propagate deletes from one site to another.

5 Summary

I have described a small set of tools that can be used, in certain circumstances, to maintain consistency between a master set of files kept on a

[9] There is middle ground. The "fetch only mode" in AFS [8] is a step in the direction of my work. In fetch-only mode fetches are made but the log is not replayed until a later time. Since log replay is what detects (and maybe resolves) conflicts, not doing it results in operation similar to mine of fetching on demand but not checking for conflicts until a locus change.

server and copies kept on a partially disconnected client.[10] Because only limited server-side system administration changes are required, users can decide individually whether to use the toolkit.

The work makes two main contributions. First, the software enables partially connected users to access their NFS home directories from stock UNIX clients. Second, the toolkit design provides a stark contrast to earlier work on custom file systems for partially connected operation. The toolkit is not as fast, not as general, and may depend on outside forces to maintain the validity of key assumptions; however, the implementation at user level makes the work more portable and provides many degrees of freedom in selecting how the toolkit operates.

A partial list of "such degrees of freedom" is: when to perform checksums, whether to test for violation of the single-locus-of-updates assumption, how large is a "large file," user-programmable cache cleanup, ability to invoke programs that automatically resolve conflicting updates, and how to manage the dialup connection.

5.1 Future Work

One obvious area for future work is difference copying. A number of commercial packages exist for Windows and provide function similar to the toolkit. Most of these packages provide for copying only differences in changed files. This is a complication, as more state must be kept on both sides. My personal experience so far has been with small text files, so the compare/copy phase runs fast enough. But in different settings differencing could be crucial.

Availability

The toolkit is available at `http://www.mcl.cs.columbia.edu/src/cfs`.

6 Acknowledgements

This work was performed while the author was in the Computer Science department at Columbia University. The work was supported in part by ONR grant number N00014-93-1-0315, and in part by DARPA order number B094, monitored under ONR contract number N00014-94-1-0719.

[10] The "disconnection model" is that connection by telephone is expensive and to be used frugally, but is always available on demand. There are other disconnection models.

References

[1] M.A. Cooper.
Overhauling Rdist for the '90s.
In *Proc. Sixth Systems Administration Conf. (LISA VI)*, USENIX, pages 175–188, October 1992.

[2] M. Crispin.
Internet Message Access Protocol – Version 4.
RFC 1730, IETF Network Working Group, December 1994.

[3] R.G. Guy et al.
Implementation of the Ficus Replicated File System.
In *Proc. Summer 1990 USENIX Conf.*, USENIX, pages 63–71, June 1990.

[4] J.S. Heidemann, T.W. Page, R.G. Guy, and G.J. Popek
Primarily Disconnected Operation: Experiences with Ficus.
In *Proc. Second Wkshp. on the Management of Replicated Data*, IEEE, pages 2–5, Novemeber 1992.

[5] J.S. Heidemann and G.J. Popek.
File-System Development with Stackable Layers.
ACM Trans. Computer Systems, 12(1):58–89, February 1994.

[6] L.B. Huston and P. Honeyman.
Disconnected Operation for AFS.
In *Proc. USENIX Mobile and Location-Independent Computing Symp.*, USENIX, pages 1–10, August 1993.

[7] L.B. Huston and P. Honeyman.
Peephole Log Optimization.
IEEE Computer Soc. Bull. Tech. Cmte. on Operating Systems and Application Environments, 7(1):25–32, Spring 1995.

[8] L.B. Huston and P. Honeyman.
Partially Connected Operation.
In *Proc. Second USENIX Symp. on Mobile and Location-Independent Computing Symp.*, USENIX, pages 91–97, April 1995.

[9] J.J. Kistler and M. Satyanarayanan.
Disconnected Operation in the Coda File System.
ACM Trans. Computer Systems, 10(1):3–25, February 1992.

[10] P. Kumar and M. Satyanarayanan.
Log-Based Directory Resolution in the Coda File System.

In *Proc. Second Intl. Conf. on Parallel and Distributed Information Systems*, IEEE, pages 202–213, January 1993.

[11] P. Kumar and M. Satyanarayanan.
Flexible and Safe Resolution of File Conflicts.
In *Proc. USENIX 1995 Tech. Conf.*, USENIX, pages 95–106, January 1995.

[12] R.G. Minnich.
The AutoCacher: A File Cache Which Operates at the NFS Level.
In *Proc. Winter 1993 USENIX Conf.*, USENIX, pages 77–83, January 1993.

[13] L.B. Mummert and M. Satyanarayanan.
Large Grain Cache Coherence for Intermittent Connectivity.
In *Proc. Summer 1994 USENIX Conf.*, USENIX, pages 279–289, June 1994.

[14] L.B. Mummert, M.R. Ebling, and M. Satyanarayanan.
Exploiting Weak Connectivity for Mobile File Access.
In *Proc. Fifteenth ACM Symp. Operating Systems Principles*, ACM, pages 143–155, Dec. 1995.

[15] B.D. Noble and M. Satyanarayanan.
An Empirical Study of a Highly Available File System.
In *Proc. 1994 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, ACM, pages 138–149, May 1994.

[16] J-S. Pendry and M.K. McKusick.
Union Mounts in 4.4BSD-Lite.
In *Proc. 1995 USENIX Tech. Conf.*, USENIX, pages 25–33, January 1995.

[17] J-S. Pendry and N. Williams.
Amd — The 4.4 BSD Automounter.
Imperial College of Science, Technology, and Medicine, London, 5.3 alpha edition, March 1991.

[18] P. Reiher et al.
Resolving File Conflicts in the Ficus File System.
In *Proc. Summer 1994 USENIX Conf.*, USENIX, pages 183–195, June 1994.

[19] R. Rivest.
The MD4 Message-Digest Algorithm.
RFC 1320, IETF Network Working Group, April 1992.

[20] R. Rivest.
The MD5 Message-Digest Algorithm.
RFC 1321, IETF Network Working Group, April 1992.

[21] M. Rose.
Post Office Protocol – Version 3.
RFC 1225, IETF Network Working Group, May 1991.

[22] H. Saran and S. Keshav.
An Empirical Evaluation of Virtual Circuit Holding Times in IP-over-ATM Networks.
In *IEEE Infocom*, IEEE, pages 1132–1140, June 1994.

[23] M. Satyanarayanan, J.J. Kistler, L.B. Mummert, M.R. Ebeling, P. Kumar, and Q. Lu.
Experience with Disconnected Operation in a Mobile Computing Environment.
In *Proc. USENIX Mobile and Location-Independent Computing Symp.*, USENIX, pages 11–28, August 1993.

[24] E. Schenk.
Diald source code.
ftp://sunsite.unc.edu/pub/Linux/system/Network/serial/diald-0.14.tar.gz

[25] E. Zadok.
Amd source code.
ftp://ftp.cs.columbia.edu/pub/amd/amd-upl102.tar.gz

[26] E. Zadok and A. Dupuy.
HLFSD: Delivering Email to Your $HOME.
In *Proc. Seventh Systems Administration Conf. (LISA VII)*, USENIX, pages 243–254, November 1993.

A Amd Operation

Amd has many features. The most obvious and commonly used is to mount remote NFS file systems on demand then later, after a period of disuse, to automatically unmount them.

The following is the sequence of relevant events showing how Amd demand-mounts the remote file system `serv:/u/user` the first time that any path on that file system (in this example, the file `/u/user/dir/file`) is accessed.

1. Before Amd is running, directory `/u` does not exist on the local disk.

2. Based on startup instructions, Amd creates `/u` then mounts itself as the NFS server servicing mount point `/u`.

```
Lookup              What happens
------              ------------
  /           return vnode for local /

  u           cross mount point into
              phony "NFS" f/s; return
              vnode that contains a
              made-up file handle
              meaningful only to Amd
              and which denotes the
              "root" of the set of
              links it is emulating

  user        return vnode for symlink
              whose value is
              "/amd/serv/u/user"

  /           return vnode for local /

 amd          return vnode for
              local /amd

 serv         return vnode for
              local /amd/serv

  u           return vnode for
              local /amd/serv/u

 user         cross mount point into
              real NFS f/s; return
              vnode for root of
              remote exported f/s

 dir          remote f/s returns
              handle for directory

 file         remote f/s returns
              handle for file
```

Figure 3: Example Name Resolution with Amd

Amd is also told at startup that whenever it must mount a remote file system it should do so within the directory `/amd`. (In this example the convention is that if server `something1` exports file system `/something2`, then it will be mounted as `/amd/something1/something2`.)

3. When someone accesses `/u/user/dir/file`, Amd is contacted during the name resolution because it is the process serving mount point `/u`. Since this is the first access to `/u/user`, Amd creates the local directories `/amd`, `/amd/serv`, `/amd/serv/u`, `/amd/serv/u/user`, and `/amd/serv/u/user/dir`, mounts the remote NFS file system `serv:/u/user` onto `/amd/serv/u/user` and thereafter emulates a symlink named `/u/user` whose value is `/amd/serv/u/user`. Since `/u/user` is an emulated path, there is no local disk data structure for it.

4. Access to `/u/user/dir/file` is then done normally by the kernel's name resolution routine.

   First, the kernel calls `getattr(u)` and discovers that `/u` is a mount point; Amd is the server mounted at that point. Second, the name resolution routine calls `getattr(user)` and is told by Amd that `user` is a symlink. Next, the name resolution routine calls `readlink(user)` and is told by Amd that the symlink's value is `/amd/serv/u/user`. At this point the original path `/u/user/dir/file` has been transformed to `/amd/serv/u/user/dir/file`. This path contains no symlinks or mount points serviced by Amd, but rather is an "ordinary" path leading to a file on a remote NFS file system; i.e., `/amd/serv/u/user` is a directory on the local disk, with remote NFS file system `serv:/u/user` mounted on it. The name resolution algorithm will cross the mount point and return an NFS handle for the remote file.

Figure 3 indicates the result of each `lookup` operation.

Because Amd is concerned only with name resolution, it implements only the `getattr`, `lookup`, `readlink`, and `readdir` NFS operations.[11]

[11] Amd also implements the unlink, rmdir, rename, and statfs operations, but these implementations are very lim-

B Autocacher Operation

The Autocacher is an old version of Amd plus some changes to implement caching of remote read-only NFS file systems onto the local disk. Autocacher emulates symlinks, like Amd, and it partially emulates files. Specifically, it implements the `read` operation.

Using the conventions of the example above, when resolving the path `/u/user/dir/file` Amd's only actions are to transform `/u/user` into a symlink with value `/amd/serv/u/user` and to mount `serv:/u/user` on the local directory `/amd/serv/u/user`. Access to `dir/file` is via the remote NFS server. Amd neither emulates the `dir/file` subpath nor makes local disk copies of `dir` and `file`; in contrast, Autocacher does both. Autocacher will create local disk copies of the directory `dir` and the file `dir/file`.

For efficiency, Autocacher will emulate reads against the file. It is common for a file to be looked up but not read; e.g., an `ls` operation. Therefore, Autocacher does not copy a file when it is first looked up, but only when it is first read. Otherwise, executing `ls` on a directory would cause the entire contents of the directory to be cached. When an uncached file is first read, the Autocacher copies the file and services all the `read` operations from the process that performed the first `lookup`. The second and later `lookup`s will be directed to the local copy.

More precisely, when Autocacher receives an NFS `lookup` operation, it reacts in one of three ways:

1. If there is a local copy of the file, Autocacher returns the file handle for the local file.

2. If there is no local copy but there is room on the local disk for the file, Autocacher returns a manufactured file handle that is meaningful only to Autocacher. A later `read` request for this file handle will cause the Autocacher to copy the file to the local disk and to implement `read` requests by reading from the local copy.

3. If there is no local copy and there is no room on the local disk for the file, Autocacher also returns a manufactured file handle, but later NFS `read` requests for this file handle will cause the Autocacher to emulate a symlink pointing to the remote file.

Just as Amd automatically unmounts recently unused file systems, Autocacher automatically deletes local cached copies of recently unused remote files.

ited, serving only to help Amd help mask its presence in the name resolution process.