File Availability and Consistency for Mobile Systems

Leeann Bent, Edward Elliott, Michael Semanko, Timothy Sherwood
Department of Computer Science and Engineering
University of California, San Diego

Abstract

One of the major difficulties in mobile computing is keeping network-based files available though the user's computer may be weakly connected or disconnected. In this paper, we survey a variety of techniques for enhancing file availability and consistency. The techniques include performing file replication, hoarding, and reintegration. Different implementations of these techniques are examined and compared. Various improvements are suggested to these schemes.[1]

1 Introduction

In traditional distributed computing, files are highly available during normal operation due to continuous connection to file servers. When network connections go down, a whole network of computers may fail; however, since these disconnections are seen as temporary, dependence on the remote server is tolerated.

In mobile computing, connection quality is more variant. Some mobile computers (i.e. home systems over a modem) are primarily connected, but connection quality is poor. Systems like this which experience intermittent, low-bandwidth, high latency, and/or high expense connections are termed *weakly connected* [6]. Other types of systems, such as traditional laptops and PDAs unplugged from the network, are primarily *disconnected*. Finally, connection costs may be so high (as with a cellular modem) or connection quality may be so poor (as with some phone lines) that systems may be forced to alternate between weakly connected and disconnected states.

The goal of mobile computing is to make less than ideal connectivity as transparent to the user as possible. Ideally, we would like the mobile user to be completely unaffected by their poor or nonexistent connection, to the point that the user would not even need to be aware that they are operating in such a state. More realistically, we would like to reduce the burden on these users by making network resources available without a large performance decrease. This transparency of computing locations can only be achieved by caching all of the needed files from the user's primary connected computing environment (including binary programs and libraries) onto his mobile computer.

With today's laptops, caching due to size limitations may be somewhat less of an issue. Current laptops have large enough hard drives to allow most of the user's primary environment to be mirrored. Smaller computing devices however, such as PDAs and smart-phones, still have limited storage space. Additionally, in some computing situations the amount of data is very large, too large to fit all of it onto even today's generous laptop hard drives.[2] As more and more information becomes available, users want to take advantage of that information, sometimes in a weakly connected or disconnected manner.

More importantly, while it may be possible to cache most or all of the files that a user needs onto his laptop or mobile computer, in many cases the user is unaware of which files they need. A perfect example of this is a user editing a document which includes a particular font. The user most likely is, and probably should remain, completely unaware that the editor needs to access a particular font file. This also happens in large, multiple-programmer projects where any given programmer may be unaware of the files needed for lower layers of the implementation. If the user encounters a missing file of this type while weakly connected, work may come to a halt until this file can be retrieved. The performance penalty for doing so may be quite severe (including an inability to retrieve the file in a disconnected state). There is no easy way to catalog all of the necessary yet obscure files the user needs, without profiling.

Additionally, users would like their local environment to reflect their primary computing environment. Caching

[1] This work was conducted under a grant from Chez Bob and the junk food provided therein. In no way are the Skittles or Snickers bars provided from aforementioned place of venue affiliated with the work presented. Should this or any other paper cause the reader physical and/or mental damage, the authors of this paper are free from possibility of prosecution in this or any other dimension.

[2] Consider the web, a very large database such as the human genome database, or an inventory of a large company for the past ten years

techniques should provide this transparency, allowing the user to work as if they were fully connected to the network.

As mobile computing becomes more prevalent, users are going to encounter more problems propagating changes to networked files back to files servers. At a minimum, users have to remember which files they have updated and copy these back to the server. More insidious problems arise when multiple users modify the same file while one or more of them are disconnected. Detection of all such write sharing conflicts can be extremely tedious if not automated, and users must still find a way to reconcile conflicts after detection.

In an ideal mobile computing environment, a user can simply grab their mobile device and work anywhere, without worrying about caching files or using a different environment. If connections are available, they can download new files and check for updated versions of cached ones, if not, they simply work disconnected. File reintegration would occur without much user intervention, either when fully reconnected to the network or periodically while weakly connected.

The properties of the ideal mobile environment can be embodied in two principles: file *availability* and file *consistency*. Availability is a measure of how accessible files are to the user, whereas consistency is a measure of how similar the files on the workstations are to those being stored at the primary location. These two qualities often conflict and a proper balance must be maintained for efficient operation.

Two techniques used to achieve file availability are *replication* and *hoarding*. Hoarding is the practice of caching shared networked files on the local machine. Replication refers to the duplication of files across multiple servers or workstations in order to decrease the cost of obtaining a file, thus increasing availability.

Consistency is achieved through the use of cache *coherency* and cache *reintegration*. In these processes, the files cached on the mobile computer are synchronized with the network files. Coherency attempts to ensure the most recent version of the file has been cached, while reintegration propagates local modifications back to the network.

Different mobile computing models require different schema for availability and consistency. In weakly connected systems, file coherency and reintegration can be implemented as background processes. The cost of reintegration must be balanced against the need to have current copies of networked files available to other users. File fetches, while costly, are not as crippling as cache misses. Users may be given the option to perform a costly fetch, depending on how critical the particular file is.

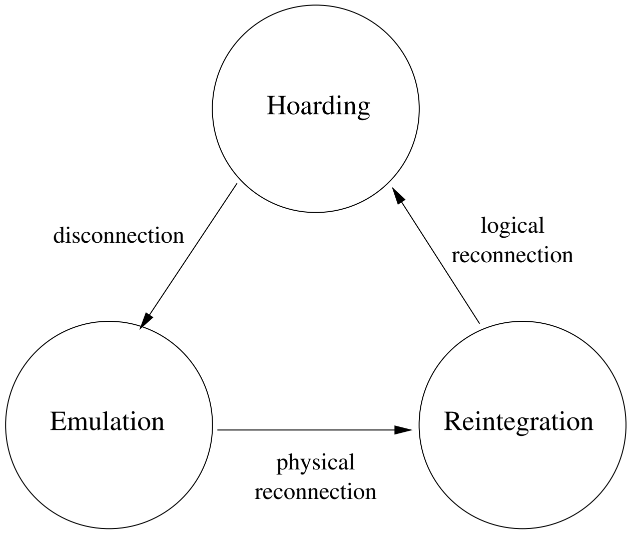

Figure 1: *Three state model introduced in the CODA work. Before disconnection the system hoards the files it needs, During disconnection it transfers to emulation mode, where it uses the local files as if it could still access the network. This is followed by reintegration on reconnection. [4]*

For primarily disconnected operation, hoarding and reintegration time is less of an issue, since users presumably initiate periods of disconnection; several minutes spent hoarding and reintegrating files is preferable to the alternatives of being unable to work or losing updates in the disconnected state. However, the user still wants as much automation in this process as possible.

In systems that experience periods of both disconnected and weakly connected operation, flexibility is of primary importance. Hoarding techniques from disconnected operation are needed, but occasional cache misses may be serviced. Depending on the connection quality, some background reintegration techniques can be used, or fetch only modes of operation to service cache misses can be provided.

This paper is organized in the following way. Section 2 deals with the issue of file availability and is divided up into a discussion of techniques for replication (section 2.1) and hoarding (section 2.2). A section 3 then considers reintegration as it applies to mobile computing. Conclusion and further discussion then follows in section 4.

2 File Availability

One key issue issues in mobile computing is file availability. To be usable, mobile computers must have the files the user needs available at all times. There are two general methods for ensuring file availability, file replication and hoarding.

Replication, used in weakly connected or disconnected systems, is built upon a more connected model than hoarding. The goal of replication is to create several copies of a file, and keep those as available and consistent as possible given a connection constraint.

In hoarding, the system assumes a disconnected mode of operation, and attempts to locally gather enough files to maintain constant availability. Consistency is postponed more than in replication systems.

Some of the different implementations of replication and integration are discussed below. Many of them are taken from larger projects that include several schemes. Each of these implementations has advantages and disadvantages which are discussed both here and in the conclusion.

2.1 Replication

Replication is the practice of storing multiple copies of a file in a file system. Generally, replication increases file availability and, in weakly connected systems, may decrease file latency.

Within replication there are several types of availability models. These include one-copy availability, primary copy availability, majority consensus, weighted voting, and quorum consensus [2]. Most models evaluated use *one-copy availability*, which is the premise that one copy of a file should always be available for editing. In contrast *primary copy availability*, which guarantees only one copy is available for systemwide use.

In addition, these one copy-availability models are managed in several different ways, the two most important being *primary/secondary management* and *peer-to-peer management*. Primary/secondary management makes one server primarily responsible for the management, availability and consistency of a file. Peer-to-peer management makes no one server responsible for a particular file, and allows all replicas to be considered equivalent in terms of availability and consistency.

Several different types and implementations are presented below. File locking is an example of primary copy availability, in which only one copy of a file is allowed to be edited at any one time. Two types of primary/secondary managers are discussed, Coda [4], which distinguishes between classes of objects, and Ficus [2], which distinguishes between a file and it's machine implementation. Both of these systems use one copy availability. Finally, dynamic sets are discussed as a general mechanism to improve access time in all availability models.

2.1.1 Locking

Ideally to preserve consistency of a file, there should be only one replica of any file. This file could be sequentially updated by all of the writers who wish access to the file.

Gray and Cheriton in "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency" suggest a method of primary copy availability which may be useful during disconnected operation[1]. To avoid reintegration, they suggest that file rights be given out to clients in such a way as to protect the coherency of the data. This leads to problem due to the fact that a client could take rights on files and hold them indefinitely. This in turn would make it impossible for other clients to use that data. To circumvent this, they introduce the idea of *leases*.

Instead of simply giving file rights out to clients, file rights are leased out by a server. A lease in this context means that the client may have rights to the file for the *term*, or duration, of the lease, after which time the client must resynchronize the data and return the rights.

Leases are useful because they eliminate the need for a tricky reintegration phase; however, they raise some interesting problems as well. First, the terms of the lease must be chosen to prevent short terms from hindering useful work and long terms from locking the system. A second issue that comes into play during disconnected operation is clock synchronization. A term is an amount of time as the server sees it, but there may be skew between clocks on systems. This would cause the client to believe it has more time to work with a file than it actually has. This might not be a problem as long as skew is kept low and the term lengths have appropriate buffer zones.

2.1.2 Primary and Secondary

While leasing allows only one writer per file, primary/secondary schemes allow many users to write to a file. This is accomplished by creating special physical or logical sets of files that can be edited by any user and synchronizing their writes. Coda and Ficus both use variants of this scheme to allow editing of local copies when operating in disconnected mode.

In the work done by Kistler and Satyanarayanan in "Disconnected Operation in the Coda File System" [4], Kistler and Satyanarayanan propose that replicas be divided up into two categories, *first class* and *second class*. First class replicas are files of "higher quality." This means that they are more persistent and trustable. Second class copies have degraded or unknown quality and are only of use if they are periodically synchronized with the first class copies (see Section 3.1).

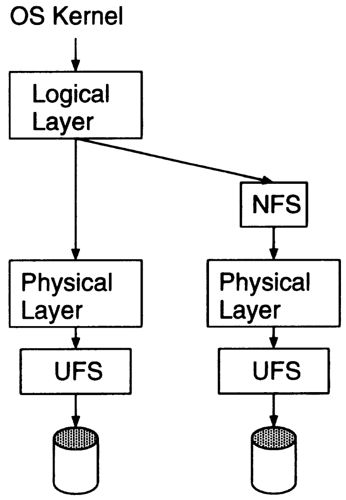

Figure 2: *The logical and physical layers in Ficus. Replication can be performed over various file systems using the logical/physical concept.*

The idea behind having first and second class replicas is that second class copies are both faster and more accessible. The second class copies, once obtained, can be used in any manner, even during periods of disconnection, because they become local to the machine. This optimistic approach of letting everyone do what they please to local copies works quite well as long as the amount of write contention for files is low. In the system that they studied (mostly a research/coding environment) they found that the amount of write contention was only 0.4%.

Ficus uses a primary/secondary scheme that keeps the difference between first class and second class replicas transparent to the user. Replication in Ficus is managed in two layers. The first layer is the abstract *Ficus logical layer*. This logical layer manages a set of replicas, called the *Ficus physical layer*, in the Ficus system (see Figure 2). These replicas are identical or slightly modified versions of the file represented by the logical layer.

The logical layer does all of the bookkeeping for the physical layer, and is responsible for concurrency control, replica selection, update propagation, and automatic reconciliation of directories. When an update is done to a Ficus physical layer file, the invoking logical layer notifies it's other physical files that an update has occurred. Each of these physical files contains a new version cache, which stores update information until the physical layer can actually do the update. Thus, the primary (in this case the Ficus logical layer) controls access and updates to the secondaries (the Ficus physical layer).

While the abstraction of logical and physical layers is a good one, there are a couple problems with this scheme. The most obvious is that since the logical layer is responsible for noticing and propagating updates, this method loses much of the availability gained by the replication. Should the machine the logical layer resides on go down, there is no way to propagate and reconcile files.

The layering technique of Ficus has the potential to slow down machine operation significantly. The authors initially displayed some concern about this; however, they claim that Ficus is fast enough in most cases.

The idea of locally caching file updates until each machine is ready to integrate them is a sound one. A busy machine need not update until it finds spare cycles. A caching service of writes propagating up to the server might also be helpful, however, to allow machines flexibility in committing their changes. This would also easily allow disconnected operation, since writes are cached until reconnection occurs. Ficus was later ported to disconnected operation, and one must assume this is how it is implemented.

2.1.3 Peer-to-Peer

Peer-to-peer schemes offer most of the advantages of primary/secondary schemes while requiring less connectivity than primary/secondary schemes. In peer-to-peer schemes, regular updates are propagated through a peer to peer connection. This scheme may take a little longer to propagate updates, however the bandwidth decrease offsets this slight disadvantage.

A good example of peer-to-peer management of replicas is Bayou [10]. Bayou is designed with both weakly connected and disconnected machine models in mind. Because of this, Bayou implements a one-copy availability model and a peer-to-peer update/reconciliation protocol. This allows the system to maximize its bandwidth and propagate updates as cheaply as possible. This scenario also allows most application clients to also function as replica servers.

Writes are propagated through the Bayou system during *anti-entropy sessions*. In these sessions two servers connect and come to a mutual agreement as to the set of write operations that they have seen and the ordering of those writes. Note that even on conflict, files are still accessible. This is a contrast to the locking mechanism of Ficus, which allows no updates or accesses after a conflict has been detected. This policy guarantees even more availability.

Because of the need to eventually commit to some consistency within the systems, however, Bayou resorts to a version of primary copy management, utilizing a primary commit server to eventually commit updates. All writes before the timestamp are committed, while those after remain tentative. Thus, unless the anti-entropy session is done with the commit server, those writes exchanged during the session remain tentative. This mechanism allows tentative writes to be rolled back and re-executed in proper order to guarantee consistency of all the replicas.

One interesting detail is that Bayou allows for protected anti-entropy sessions using public key protection on files. While this is a good idea from a security standpoint, it increases the overhead of tentative writes by a significant factor. Storing this information may or may not be beneficial, since anti-entropy sessions would usually happen between trusted parties.

Most features of the Bayou availability model seem well suited to mobile computing, especially the anti-entropy sessions. An issue that isn't explored is what would happen should the primary commit server go down permanently. Since tentative writes take up more space than committed writes, this might be an issue. Some potential solutions to this are to allow redesignation of a primary server, or to force commit of tentative operations within a certain time period.

2.1.4 Dynamic Sets

The above schemes are all designed to reduce the amount of work required to ensure availability and maintain consistency within a replication environment. While each of these has advantages, there are also techniques to improve the speed of any replication scheme. One general technique, called dynamic sets, can be used to speed up file accesses and updates.

Steere and Satyanarayanan present the idea of *dynamic sets* of files [8] in order to increase the performance of file fetching. Normally, when a group of files is fetched from a disk or from another computer, the fetching is done sequentially. This can waste a lot of time overall because programs fetching the files do not consider access latencies. Dynamics sets reorder the files by latency, fetching the smallest latency file from the file server first first. This allows computation to begin much earlier while the other files are being fetched.

Dynamic sets have four major operations: `setOpen`, `setClose`, `setIterate`, and `setDigest`. The first two functions exist to allow the handling of sets as objects. `setOpen` opens a dynamic set of files either by expanding filename wildcards or by running some query. `setClose` closes the set and performs cleanup. The next two functions exist to allow the programmer to go through the files in a set and perform actions on each one. `setIterate` opens the next file in the set in order of latency to retrieve. This is the normal opening operation that would be used in the place of simple file opens in lower level programs. `setDigest` examines summary information about a file (i.e. a filename, a header, or a thumbnail) and then moves to the next file. Thus the user can quickly scan through the files using `setDigest`, an additional feature that most file systems do not support.

This method has been shown to reduce the total elapsed time on certain types of applications by almost a factor of four at 9600 baud. Because processor speed has increased so dramatically while transmission speed has not, this method would be even more effective on current systems. One thing not really discussed in the paper is that dynamic sets could also make much faster would be parallel I/O. The unordered nature of the sets could allow multiple processes to operate in parallel on all on the files much more easily than the normal serial counterpart.

In addition, the dynamic sets prototype created by Satyanarayanan has many inefficiencies that could be removed to greatly speed up operations. For instance, the prototype uses off-the-shelf RPC packages that limit the amount of data transfer. This design decision can cause an unnecessary amount of overhead when trying to transfer large files. The prototype also used a simplistic user-level thread package. The inefficiencies here arose from the blocking of any application whenever any thread makes system calls (such as those created by reading from a socket). With a more complex threading scheme, more processing could take place during these calls. With such a high performance, the elimination of these inefficiencies is not even needed to achieve high efficiency.

The dynamic sets implementation also has some problems. For instance, the set groupings themselves are static in nature. Once the query is run to retrieve the files, these files remain in the set, regardless of whether or not the files actually exist on any systems. This could make programming with dynamic sets slightly more difficult, as the program must then check to make sure that the file actually exists when it does perform the operation on it.

2.2 Hoard Techniques

Even though mobile users may periodically disconnect from the network, they would still like to be able to access shared network files. Caching these files locally provides this access, but normal cache loading policies are insufficient in disconnected and weakly connected environments. Cache misses can not be serviced in a disconnected state and may be too expensive to service in a weakly connected state. On the other hand, timing is not critical when loading the cache before the client surrenders strong connectivity; such events are usually user-initiated, and a cache loading policy which takes several minutes is acceptable.

The general hoarding problem poses the question of how to best load a client's cache with files before the client disconnects from the network. Solutions to this problem are known as *hoarding strategies*. Since the penalty for cache misses is severe in mobile environments, most strategies aim to eliminate cache misses entirely. Thus hoarding more files than the client actually uses while mobile is tolerable and probably necessary. Of course, the client seldom has space to hoard all shared files from the network, so some selection of files must occur. Manual hoarding solutions rely heavily on the user to fill the cache with needed files, while automatic solutions attempt to predict which files the user will need, using varying degrees of user input.

Several mobile computing projects have addressed the issue of hoard strategies. In the LITTLE WORK project at University of Michigan's Center for Information Technology Integration (CITI), one of the earliest projects to consider hoarding, Huston and Honeyman implemented a manual LRU cache loading policy [3]. Kistler and Satyanarayanan, in their work with Coda at Carnegie Mellon University, used a more automated cache loader, allowing users to prioritize files with a hoard profile [4]. Kuenning and Popek invented semantic distance automated hoarding for the SEER project, attempting to measure the relationships between files [5]. Finally, Tait *et al* from IBM's Watson Research Lab utilized transparent analytical spying, where file accesses from multiple program executions are analyzed [9].

2.2.1 Manual LRU

The most basic type of hoarding, involves loading those items most recently accessed into the hoard. While this process may not load the optimal working set, it is easy to implement and use. In addition, it is easy to integrate an LRU hoarding scheme into a local file system.

The LITTLE WORK project at CITI was begun to provide mobile users with access to shared resources from disconnected laptops. Rather than requiring an explicit manual copy of files, Huston and Honeyman hoped to provide a local cache manager that would store needed files automatically and provide access to these files while disconnected. Further, this caching mechanism had to work with the large AFS network already in place at CITI.

The AFS file system provides consistency guarantees to its clients, but requires the client to maintain network connectivity. The client caches files locally and is issued a callback for each file by the server. As long as the client possesses the callback, it has the most recent version of a file. When the server detects a modification to a particular file, it revokes the callback from all clients.

When a user wishes to disconnect his computer from the network, he first runs all applications he will need while disconnected. Since the cache manager removes files on a least recently used (LRU) basis, this loads the cache with the files for those applications. The user then issues an explicit disconnect command to his machine.

To maintain the existing network, changes to the AFS servers were prohibited; all changes were required to be done in the client. AFS provides a client cache manager which has pessimistic behavior: in the absence of an explicit server callback, it assumes cached data is invalid. Huston and Honeyman modified the cache manager to be more optimistic when disconnected so cached data could still be accessed when a server can not be contacted. This optimistic mode is triggered by the disconnect command.

In disconnected mode, if an operation is requested on a cached file, the operation is logged and performed on the local copy. Extra information is also logged to allow the operation to be replayed to the server upon reconnection. This propagates changes back to the file server. Cache misses in disconnected mode are handled by returning an error.

Huston and Honeyman's manual LRU hoarding strategy relieves users from needing to know which files are required to run their applications. This method has several drawbacks. Most notably, file accesses by an application can vary widely across executions. Many application functions reside in library files which are only loaded on demand. If the user does not perform each action within an application when he is hoarding files, not all libraries will be loaded and some actions will be unavailable during disconnected operation. Users must also remember which applications they intend to run when they are hoarding.

2.2.2 Hoard Profile

In contrast to the LITTLE WORK project, the CODA [4] project was born out of a need to provide users with uninterrupted service during a file server failure. They felt that providing disconnected operation was a natural extension of the facilities needed to provide fault tolerance in a distributed file system, and hence some of the mechanisms are not as efficient as possible when applied to mobile computing. It should be noted, however, that even though CODA was born from different design goals many of the approaches that they use are quite similar to LITTLE WORK.

Kistler and Satyanarayanan use a simple approach to determine which files to hoard on their system. They use a combination of what they deem *prioritized cache management* and *hoard walking*. However this is really simply a combination of a version of LRU combined with a user's fixed list of directories and files to cache. The *current priority* of a cached object is a combination of a score for it's recent usage (LRU) with it's fixed user defined hoard priority.

LITTLE WORK and CODA both share this idea of LRU, however the CODA project introduces the idea of prioritized cache management where the user specifies which files need to take priority in the cache.

Kistler and Satyanarayanan originally state in their design goals that they wish to make a transparent file system; however, the user-defined priorities are a clear step away from this. They may be able to get around this by arguing that either the profiles are manageable by an administrator or that the they are not really needed during normal operation (i.e. that the LRU metric works well enough). We can see that neither of these is true. The administrator usually has no idea which files are related to each other and general should not have access to the user's files. Later papers show that more complicated heuristics are needed than those presented.

2.2.3 Semantic Distance

The SEER [5] project takes exactly the opposite approach of the CODA work. SEER uses *automated hoarding* to increase file availability on disconnected workstations. The basic idea behind the SEER method is to arrange the files into groups with semantic meaning. All of the files that would apply to a specific task are be placed in a group. SEER does this without any required user interaction and with an order of magnitude improvement in system performance versus other approaches.

There are two main components to the SEER hoarding system: the *observer* and the *corellator*. The observer watches user file activity and notes every file access. The corellator computes the *semantic distance* between every pair of files. Semantic distance is based on the idea that two files accessed at nearly the same time are much more likely to be related than two files that are accessed at very different times. This information is then supplemented by directory membership information, naming conventions (such as `.EXE`), and "hot" links (such as `#include` in `.c` files). Then the corellator runs a clustering algorithm that attempts to put the files into overlapping task-sets based on this distance measurement. Once the files are put into groups, the system merely needs to know what groups to hoard, and then the user should hopefully have all of the files necessary to complete the user's tasks while disconnected.

While the numbers in this paper show a significant improvement over previous methods, such as LRU, there are many additional hacks which have been used to improve SEER's performance. These improvements stray from the original design of being able to determine which files are related in a general fashion. For instance, shared libraries caused a major problem with the clustering algorithm, as they cause everything that used the shared library to be clustered together. Due to system libraries, the waste of space on the workstations by unused programs became completely unacceptable. This was fixed by making such shared libraries (and all other frequently accessed files) always included in the hoard, and removing them from clustering computations. Also included in the hoard was every file that began with a period (for instance `.cshrc`). These files have the property that they are almost always small, critical files; however, it is unfortunate that the general algorithm could not determine this for itself rather than relying on explicit specification. Kuenning and Popek have no statistics for how the system would have run without the multitude of special cases that had to be added for "real world intrusions." This is probably because the system would not work without these additions.

While the SEER project looked carefully at various ways to define the semantic distance between two files, they accepted their complicated clustering algorithm without so much as a discussion of other techniques for clustering. It is in the clustering that many of their problems arise. For instance, normal clustering does not take into account directed relationships. The fact that one file was opened before another can be an extremely useful piece of information. When a shared library is used, for instance, the application is opened, and then the library is opened. Under another algorithm, this might have indicated a directed relationship that would include the library if the application were needed, but would not include the application if the library were needed. This is also much more general than the frequently accessed file technique.

2.2.4 Transparent Analytical Spying

Transparent analytical spying [9] is a generalization of LRU hoarding that incorporates heuristics similar to those used in the SEER project. These heuristics are used to create relationships between different files. While those heuristics used in SEER are very specific, transparent analytical spying generalizes these somewhat.

Given the basic hoarding problem, Tait *et al* wanted to devise a scheme to intelligently fill the disconnected user's cache. They hoped to design a hoarder which could be effectively used by both novice and advanced users.

The transparent analytical spying approach to file hoarding uses file working sets to load the cache. Users setup *book-*

ends to delimit spying periods, during which file accesses are detected. When the user wants to load the cache, the bookends are analyzed and the user is presented with a list of programs to hoard. Merging the results of several bookends helps gather all files needed by a program, not just those encountered in a single execution.

During spying, network file accesses are recorded along with the pid of the process making the access. The log analyzer, run at hoard time, creates a tree from the log information, with each file as a node. Children of a node are files opened or executed by the parent. The root of the tree is the shell, and its immediate children are the roots of program trees. Thus each subtree of the the root node represents a single program execution. This allows files to be associated with specific applications. (See figure 3

To separate user from application data, the authors employ several heuristics. OS/2 recognizes naming conventions similar to DOS, so in the first step filename extensions are analyzed. If this fails to distinguish the file type, the analyzer makes inferences from the file's location. If a file does not share at least the top-level directory with its parent program, it is assumed to be data. Finally, modification timestamps are used as a last effort. Program files are usually installed once and not modified, whereas data files have usually been modified frequently and recently. Thus child and parent timestamps are compared to determine the child's status. Since these heuristics sometimes fail, the default behavior is to hoard all files in an execution tree.

The shared network files and local client files are stored on different file systems. Due to the vagaries of OS/2, it is only possible to trace network file accesses. This creates an *orphan problem* for the hoard analyzer. *Data orphans* result when a locally installed program accesses data files on a network server; these data files can not be associated with any program. A remote program (that is, a program stored remotely, not one executed remotely) which accesses only local data becomes a program orphan. *Program orphans* are automatically loaded at hoard time, while data orphans are presented to the user.

Merging trees from different program executions provides a picture of all files accessed by that program, but loses information on how frequently each file is accessed. This information could help make decisions when cache space is limited on which files to hoard. If each node in an execution tree included a count field, the number of accesses of a particular file across executions could be recorded. As the trees are merged, the count could be incremented each time a duplicate node is encountered. This would also allow the number of times a particular file is accessed during a single program execution to be counted.

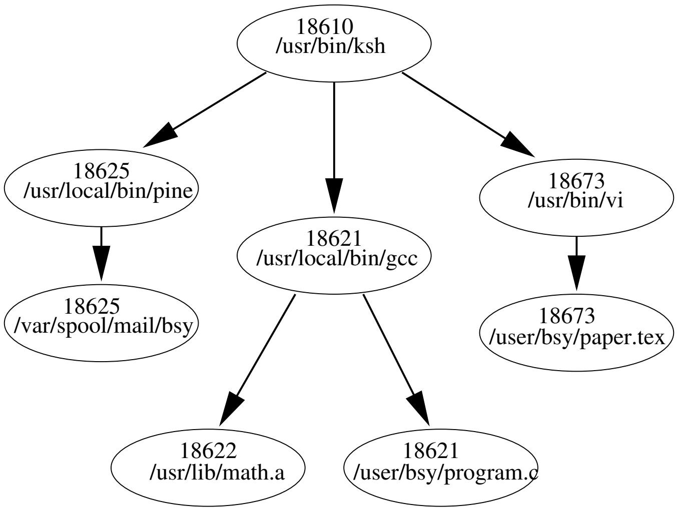

Figure 3: Sample file access tree reconstructed from log file

3 File Consistency

Mobile users benefit just from having copies of shared network files when disconnected or weakly connected. However, users would like to be able to modify files while mobile and have updates propagated back to the file servers. Mobile clients typically undergo a reintegration phase upon reconnection to the network where they reconcile modified cache data with network copies. This raises several sharing issues. Write conflicts occur when a connected and a mobile user both write to the same file. Read conflicts can occur when a mobile user reads a file from its cache which is out of date. Effectively handling these conflicts is a major goal of mobile computing.

Scheduling reintegration entails two competing issues. On one hand, reintegration can be a lengthy process, and mobile clients may want to delay it as long as possible. On the other hand, the longer a file write remains unknown to the server, the greater the chance a write conflict arises. Mobile clients are also typically at risk for damage, theft, or loss. Weak connectivity can often be utilized to mediate these interests.

Conflict resolution techniques vary from system to system. The simplest solutions write conflicting updates to new files and notify the user. More advanced techniques attempt to reconcile conflicts without user intervention. While some mobile clients attempt to perform all the reintegration work themselves, more elegant solutions involve file servers in the process. In fact, many of the conflict resolution techniques of replicated file systems greatly ease the burden on the mobile client. For replicated systems which allow multiple concurrent writes, a mobile user reintegrating data poses few new problems.

Huston and Honeyman's LITTLE WORK project at CITI delayed all reintegration until full network reconnection [3]. Mummert and Satyanarayanan's work on Coda at Carnegie Mellon introduced large granularity cache coherence [7]. Mummert, Ebling, and Satyanarayanan exploited this idea in their efforts on the Coda project by using weak connectivity to perform trickle reintegration [6]. Terry *et al*'s contributions to Bayou at Xerox allow for programmable conflict resolution [10].

3.1 Manual Conflict Resolution

Kistler and Satyanarayanan present arguments for manual conflict resolution[3] in their paper "Disconnected Operation in the Coda File System" [4]. Their argument is the same mantra we have been hearing for years, make the common case fast. They find that the common case was everything except *write sharing*. Write sharing refers to allowing two people to both have write access and write to a file at the same time. In fact, write sharing is so uncommon that it only occurs 0.4% of the time on their measured software development workload. As such they implement an optimistic coherency and reintegration technique where they simply assume that every write is non-conflicting.

At reintegration time, they simply look for those files with two or more conflicting cached copies. When this occurs the reintegration method fails, leaving the users to sort out the details.

This method works quite well when there is a low degree of write sharing, such as in a research oriented software development setting; however, it is yet to be seen how universal this policy is. There will be problems as write conflicts increase to any significant proportion of file operations.

Huston and Honeyman implemented a similar policy in their efforts with LITTLE WORK [3]. When the client reconnects and replays its log file, several conflicts may arise. AFS files have a timestamp and a version number, which increments every time a file is modified. On replay, local files version numbers are checked against the server's version numbers. If a connected client and disconnected client both modify the same file, the replay agent informs the user that a conflict has occurred. The locally modified version of the file is stored on the file server with a different name, and the user must resolve the conflict.

Because AFS guarantees connected clients always read the most recent version of a file, a disconnected read can also cause a conflict. If the local and server version numbers conflict during replay of a file read, the user is informed that he may have read stale data. Because such checks require synchronicity of client and server time clocks, which cannot be provided, a window of time is considered instead of single instants.

Problems arise when a file or directory is deleted from the server while a disconnected client modified that object. When replaying the log, if a file or directory cannot be modified on the server in its original location, it is placed in the orphanage. This also occurs when permissions on the server don't allow the client to modify an object. In these cases, the user is notified that his object was saved in the orphanage directory.

Replaying a log file can be a lengthy process. If the user needs to reconnect to the network to obtain a file not in the cache, but doesn't have the time or bandwidth to replay a large log file, he can run in "fetch-only" mode. In this mode, cache misses are serviced. Operations which don't modify files are passed to the server while other operations are logged.

Similar to Coda and LITTLE WORK, the reconciliation algorithm in Ficus [2] is very rudimentary. Reintegration is handled by simply copying over the most recent of the update (indicated by a dominant version vector). If neither file's version vector is dominant, Ficus fails, and the conflict is recorded to be arbitrated by the user. Normal access to any set of files with a conflict is subsequently blocked. Directory reconciliation is done by simply unioning the updates, and subtracting out the deletions. Name conflicts within directories are handled by saving both files, and assigning a disambiguating file extension.

The conflict resolution mechanism in Ficus is poorly implemented. Rather than attempting to reconcile files to retain all writes, the most recent version is used. While non-dominant writes are not overwritten, neither of the Ficus papers discuss whether this means that concurrent, disconnected updates are non-dominant. Directory updates are also handled poorly, doing essentially no reconciliation, and leaving all conflicting entries to the user. In spite of the fact that most issues are punted on, however, reconciliation still takes an extraordinary amount of time (45 min. over a modem line [**?**]). Finally, the fact that files lock on conflict leads to an extremely low availability, since the person needing to update may not be the same person who is resolving files. Once Ficus was ported to disconnected operation, time based reconciliation was used, instead of version vectors, so that consistency was only checked for files that were actually modified.

3.2 Hierarchical Cache Coherence

While reintegrations primary goal is thoroughness of automation, another goal of reintegration is speed. Users do not want to wait for reintegration, especially when they are

[3] also known as snarf on conflict among some circles

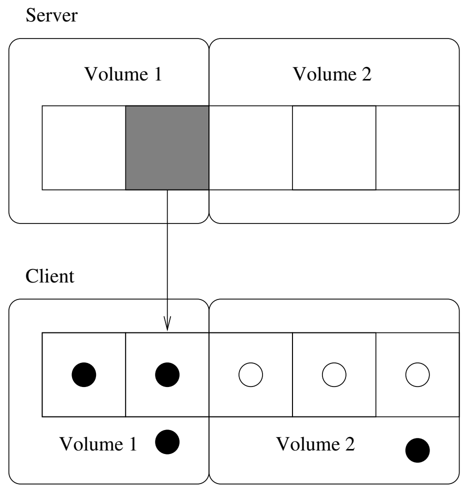

Figure 4: *The server has a newer version of a single file to provide the client. The use of volume callbacks allows fewer consistency checks to update the client. Filled in circles represent checks that had to be made to ensure cache consistency. Note that normally, each file would have to be checked. For large volumes without files that need to be updated (application files for instance), the user can experience significant savings.*

connecting and disconnecting fairly often. One of the primary optimizations for reintegration, volume level reintegration is highlighted in Coda.

One of the better techniques for optimizing callback-based reintegration is the use of *large granularity cache coherence* [7]. This algorithm by Mummert and Satyanarayanan decreases the amount of data that needs to be transmitted to confirm that a cache is still valid. After the server provides a client with callbacks for each of its files, the server will also provide a callback for the volumes that the client has. In the case that any of the files in these volumes change, the callback will be revoked. This means that the system does not need to check every file to see if the cache is valid, but rather only needs to check the volume callbacks. If the volume callback has been revoked, then it must check every file within that volume in order to make sure that the volume is consistent (see Figure 4).

This optimization decreases the cache validation time by a factor of five to twenty-five in most cases; however, this approach adds significant complexity to the reintegration process. One problem added is known as *false sharing*. False sharing occurs when the client system has hoarded some but not all of the files in a volume. A change to those files on the server would then cause the volume callback to be revoked. The client would then have to waste time to regain the volume callback before it could proceed. Although this is not fully examined in the paper, the efficiency provided by large granularity cache coherence outweighs the difficulties caused by false sharing.

For simplicity of implementation, Mummert and Satyanarayanan choose a two-level hierarchy for cache granularity: files and volumes. It is certainly conceivable that they could use a more hierarchical structure. This hierarchy need not even be based upon how the files are actually stored. Volumes can actually be rather large collections of files, and subdividing the volumes arbitrarily might allow the system to get away with checking far less files in the case of a callback-break. More levels of hierarchy would also help take care of the issue of false sharing by not returning callback-breaks on subsections of files that are not hoarded.

3.3 Trickle Reintegration

For machines that are always weakly connected, different schemes are feasible for reintegration. Specifically, the system can reintegrate while the user is working, maintaining the illusion that the user is fully connected.

Mummert, Ebling, and Satyanarayanan wanted to exploit this weak connectivity in the Coda project to improve the performance of a mobile client [6]. When weakly connected, cache coherency can be maintained using large granularity callbacks. Trickle reintegration slowly propagates updates back to servers in the background. Cache misses can be serviced, but Venus only does so if the file can be retrieved quickly. A weakly connected client using these techniques is said to be in write disconnected mode.

A large granularity in cache coherence helps take advantage of write disconnected mode by reducing the number of callback requests (see Section 3.2). Version stamps were kept for entire volumes as well as files; when any file in a volume changed, the volume's version stamp was updated. To check cache consistency, the client first requests version stamps for all the volumes which it has files from. If a volume's stamp has not changed, there is no need to check stamps for any of the files in that volume. At the expense of one extra RPC call to request volume stamps, many individual calls to request file stamps can be eliminated.

During trickle reintegration, file updates are still logged by the Venus cache manager. Sometimes one file operation undoes the effects of previous operations, as when a file creation and write are later cancelled by a delete. Venus can detect such situations and remove voided operations from the log file. Log optimizations thus reduce network traffic, a primary concern during weak connectivity. To promote log optimization, an aging window only allows reintegration of log entries after they have been in the log a certain period of time. The window only marks entries as eligible for reintegration; a daemon process does the actual work of reintegrating these entries when convenient.

Records are reintegrated in chunks. The choice of chunk size varies with network bandwidth, and defaults to 30 seconds worth of data at the current speed. This ensures that reintegration doesn't tie up the network for long periods, delaying a high priority event such as servicing a cache miss.

Cache misses which occur during trickle reintegration are sometimes serviced and sometimes not. Users are usually willing to wait a few seconds to retrieve a small file, but would rather do without a large file than wait a long time to retrieve it. The time to retrieve a file under current bandwidth conditions is computed and compared to a patience threshold value, representing the amount of time a user will wait for a file. If the value can be retrieved under the threshold time, the cache miss is serviced; otherwise, Venus returns a cache miss error.

Venus periodically performs hoard walks to check cache consistency. This occurs in two phases, a status walk to determine which files are inconsistent and a data walk to fetch those files. Between the status and data walks, the user is presented with a list of files to be fetched, and he can manually select which files to retrieve. Again, the file retrieval time is compared to the user's patience threshold. The default is to retrieve files which can be serviced under the threshold and leave all others.

Certain files are more critical to the user than others. Letting the user specify separate patience thresholds for each file would allow retrieval that is more responsive to the user's needs. Threshold values need only be kept for files the user specifies, and a default threshold used for all others. To speed servicing, a file retrieval time can first be compared to the default threshold; if the time is too high, the cache manager could lookup the file name in a small table to see if it has its own threshold value.

Volume callbacks are most effective when a volume contains files which are seldom updated. They perform poorly (although no worse than file-by-file callbacks) when they contain a large number of files, of which only a few are regularly updated. This suggests that volumes expected to contain frequently modified files should be kept small. To maximize the benefit to mobile clients, the server could periodically run an algorithm to reorganize volumes, grouping frequently modified files together. Since volume subdivisions of the directory tree are transparent to the file system, this would not alter a file's absolute path name. The server would have to be certain no clients were currently using the volumes to be reorganized, or a mechanism to resolve conflicting volume information would have to be provided.

3.4 Programmable Conflict Resolution

Bayou's reintegration model is not continuous process like trickle reintegration. Rather reintegration is done on a peer-to-peer per write basis, where these peer-to-peer meetings are assumed to happen often. Thus, reintegration happens every time two computers that share a file meet. This process works well for intermittent connectivity. Additionally, when two computers are connected, reintegration takes place on a per write basis.

One of Bayou's design goals was to make conflict resolution non-transparent to the implementor of a replicated system. To this end they created application specific conflict detection and resolution procedures. These two pieces of the system are referred to as dependency checks and merge procedures. Since conflicts are different for most application types (i.e. schedules, bibliographic references, files, etc), this allows reconciliation to be more specific, and more automated once the procedures are in place.

Dependency checks, or consistency checks, check the data integrity on write. This is the equivalent of checking a timestamp or version vector in another system. Dependency checks allow more flexibility, however, since not every update to an object creates a potential conflict in some systems.

Reconciliation algorithms, or merge procedures, allow the user to specify operations like second-best-updates (in meeting room reservation for example). Since merge procedures are only done when dependency checks indicate a problem, they have the potential to be more efficient than conflict resolution on a per file basis. Also, since system designers implement reconciliation, those who understand the system best have the ability to tailor reconciliation to their needs. Note that when no reconciliation algorithms are specified, the system reverts back to manual conflict resolution.

Eventual consistency is maintained by requiring that the merge procedures change only internal program data. Thus, if writes are executed in order on different machines with the same data, the two copies of the data will be consistent at the end of the writes.

Dependency checks and merge procedures abstract the conflicts and resolution from the mechanism to implement detection and reintegration. Some of these methods may be rather hard to implement should application programmers decide not to include predefined resolution mechanisms. Additionally, the restrictions on conflict resolution (i.e. only operations on local data), may lead to problems with the merge procedures. The efficiency and flexibility of this system more than make up for these issues.

4 Conclusion

As wireless communication continues to increase in popularity and clients become more and more mobile, file system support for weakly and disconnected operation becomes inevitable. It is with this in mind that we have presented techniques for replication, hoarding, and reintegration. Together, these techniques ensure availability and consistency over weakly connected and disconnected file systems.

4.1 Replication

Replication is an important aspect of mobile computing for several reasons. Replication is used to keep versions of a file accessible to both sides of a disconnection. Additionally, many hoarding techniques rely on replication schemes to maintain consistency of their hoards. Finally, file systems may use replication to increase the speed of reintegration for disconnected clients. Of the various replication schemes that we have seen the one that holds the most promise in the author's eyes is the peer-to-peer scheme.

There are several reasons why a peer-to-peer scheme is the most beneficial scheme seen here; however, the primary reason is that it offers the most availability for the least bandwidth cost. In fact, Ficus later switched to a peer-to-peer scheme.

The peer-to-peer scheme offers one copy availability. Since the primary goal of any replication scheme is availability, one copy availability is required of any replication scheme that is going to be disconnected or weakly connected for any length of time. Additionally, files are never unavailable in peer-to-peer replication because files are not locked on conflict (since conflicting files are seen as a normal state of the machine). Other systems do not allow writes on conflict, and some systems, like leasing, only allow one writer per file. These all decrease the availability of a file, and are counter to the purpose of replication.

Peer-to-peer replication also allows replica reintegration to be done cheaply, both in terms of network cost and in terms of time spent reintegrating. Pair-wise updates allow changes to propagate through a system without requiring that every server connect to every client for each update. In a primary/secondary scheme the server makes n connections, while the clients make only a single connection. In a peer-to-peer system it's possible for the server to make only one connection to propagate, while the clients may make anywhere from one to n connections, propagating updates amongst themselves. Thus, the peer to peer update creates fewer bottlenecks, is more reliable, and is more flexible. Additionally, because these updates have less overhead and can be done more often, each reintegration will take less time, since each model will be more consistent.

As in any model, there is room for improvement in a peer-to-peer scheme. As the authors stated earlier, the reliance on a primary timestamp to commit a file is in some ways limiting. Two methods to get around this are redesignation of a primary and timeout of files. Both of these methods are stopgap and any replication system requires a way to commit files. However, a rollback scheme may require too much work to be implemented efficiently and on a wide scale. It may also not be necessary, assuming different integration techniques from those in Bayou are used.

Dynamic sets, for instance, provides a structure which can greatly improve the performance of replication techniques. Reducing the the average latency to access a file can create great improvements when the files are distributed over the network. In the peer-to-peer scheme, for instance, the dynamic set algorithm would find the server that minimizes access latency for each file in the set, and then perform the file accesses in an order that would reduce overall access time.

4.2 Hoarding

Hoarding techniques are built primarily with disconnected activity in mind. While some of these methods utilize replication in various fashions, there is a subset of problems unique to hoarding. As we have seen though this survey, some of the problems of hoarding are very difficult indeed. Eventually, each of the systems reviewed reverted to some sort of user assisted hoarding mechanism whether it is the user profiles in CODA [4], Tait et. al. [9] and there bookends, or the hand coded #include finding mechanism in SEER [5].

There is no hoarding scheme that seems to stands out as best. However, since transparent analytical spying is a generalization of semantic distance measure, used in conjunction with bookends and user intervention, this technique is the most thorough. Hoard profiles allow the user to specify files which may be vitally important to system use, even though they may not be covered by the bookended time period. Semantic distance is a useful tool for determining file relationships, and hoarding sets of applications; however, it is not useful without some augmentation. Transparent analytical spying uses each of these sub-methods to combat the the deficiencies of the others, while passing some flexibility on to the user.

While transparent analytical spying is the most general technique, additional flexibility could be added using system specific modules. In this manner semantic distance heuristics could be plugged into the system to handle the common cases for any particular operating system or application. Also, a hoard session could save user specifications into a hoard profile for the next use. This way advanced users could specify files they need only once. Additionally, incorporating frequency of operations into the heuristics might be beneficial.

Hoarding techniques could be used in conjunction with replication techniques by using hoarding to "load" mobile clients, or to pick which subset of files to replicate across various machines. Since replication is meant to increase file reintegration ease, as well as maintain availability, hoarding could be used not only at the client level (as it is in some of the schemes we have examined, such as Coda), but it could also be used to hoard files at a local file server, from which updates to clients are gained.

Server file hoarding techniques could be used in replication schemes. A peer-to-peer replication could choose which files to store at every machine based on local hoarding techniques. While every machine would not have every file, making replication methods run slowly, this would allow a very heterogeneous environment. Additionally, global servers could merge hoard profiles or working sets created by hoarding methodologies to gather and maintain copies of needed files. Similarly, hoarding techniques could be used in primary/secondary replication schemes to determine the secondary machines for each file. This would require a standard format for hoard specifiers since each client could potentially be using different hoarding techniques when connected.

4.3 Reintegration

Because the reintegration in both replication and hoarding schemes are so similar, general reintegration techniques can be used in either, with only a few reintegration techniques dependent on implementation. Indeed, in these cases, methods of reintegration seem more dependent on connectivity than on whether a system is reintegrating due to hoarding or due to replication.

Several of the systems we reviewed use manual conflict resolution, which is clearly the most robust scheme at the experience of convenience. In is also uncertain how much can be done to improve this scheme. In any reintegration scheme, there will inevitably be certain situations that require user intervention. Automation is unlikely to decrease the number of these noticeably, except in specific application domains, due to the fact that reintegration is so content dependent. Those automated reintegration techniques that are the most successful are those that take content into consideration, for example programmable conflict resolution. Outside of this the most a reintegration techniques can hope to do is aid the user in their manual reintegration decisions.

There are many techniques used to ease reintegration, including giving hints for manual reintegration, speeding reintegration, and making reintegration necessary less often. Certain fail on conflict techniques offer guideposts for the user to reintegrate instead of helping to reintegrate. For example, LITTLE WORK offers stale data information. Also, most systems reviewed keep log files for user inspection. Hierarchical cache coherence can be used in any scheme to speed integration for the user. In weakly connected systems techniques like trickle integration offer more consistency, thus there are fewer conflicts, even though those conflict may be dealt with using a manual scheme. Meanwhile, application specific reintegration can ease reintegration a large amount for certain applications. For example, in individualized computing environments, (such as database applications, or scheduling applications) the Bayou system offers the most flexibility, and, potentially, the best and most efficient reintegration.

These reintegration techniques could be used with either replication techniques or hoarding techniques, since their functionalities are orthogonal. Various trade-offs would need to be made for some types of systems. For example, the necessary user intervention and file locking of manual reintegration might be an issue in systems meant to require less user interaction. Additionally, some of these schemes could be used together, i.e. trickle reintegration with programmable conflict detection and resolution.

It seems that the application domain for these methods is more constrained by connectivity than by implementation of replication or hoarding. Trickle integration and programmable reintegration seem to be more suited to a more connected system, as they depend on fewer conflicts, and easy to resolve conflicts. Under this system, peer-to-peer replication, primary/secondary replication, all methods of hoarding, and all methods of reintegration would be feasible. Manual conflict resolution and its various incarnations meanwhile, are a more robust schema, where periods of disconnection don't matter. Problems are dealt with on a case by case basis. Peer-to-peer replication seems to offer advantages over primary/secondary replication; however, all hoarding schemes are applicable, as are any of the reintegration schemes. In general, these methods taken together can be used to create a robust, efficient, highly available, consistent file system.

4.4 Final Thoughts

A mobile file system must provide for the availability and consistency of the files within the system. Availability, as we have seen is maintained through different copies of a file, i.e. replication or hoarding. Consistency is maintained through coherence strategies and reintegration. Because of the different models for mobile computing, however, different techniques may be used in different situations. Additionally, used together these methods provide a suite of tools with which the mobile user is armed.

References

[1] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th ACM Symposium on Operating System Principles*, December 1989.

[2] Richard D. Guy, John Heidemann, Wai Mak, Thomas W. Page, Gerald J. Popek, and Dieter Rothmeiner. Implementation of the ficus replicated file system. In *Summer USENIX Conference*, Anaheim, California, US, 1990.

[3] J. S. Heidemann, T. W. Page, R. G. Guy, and G. J. Popek. Primarily disconnected operation: Experience with ficus. In *2nd Workshop on the Management of Replicated Data*, pages 9–12, 1992.

[4] Larry B. Huston and Peter Honeyman. Disconnected operation for AFS. In *USENIX Symposium Mobile Location Independent Computing*, pages 1–10, Cambridge, US, August 1993.

[5] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, US, 1991. http://www.cs.cmu.edu/afs/cs/project/coda/Web/docs-coda.html.

[6] G. Kuenning and Gerald J. Popek. Automated hoarding for mobile computers. In *Sixteen ACM Symposium on Operating Systems Principles*, pages 264–275, Saint Malo, France, October 1997. http://fmg-www.cs.ucla.edu/geoff/sosp97.html.

[7] Lily B. Mummert, R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, USA, December 1995. http://www.cs.cmu.edu/afs/cs/project/coda/Web/docs-coda.html.

[8] Lily B. Mummert and Mahadev Satyanarayanan. Large granularity cache coherence for intermittent connectivity. Technical report, Carnegie Mellon University, April 1994.

[9] David Steere and M. Satyanarayanan. A case for dynamic sets in operating systems. Technical Report CMU-CS-94-216, Carnegie Mellon University, November 1994. http://www.cs.cmu.edu/afs/cs/project/coda/Web/coda.html.

[10] Carl Tait, Hui Lei, Swarup Acharya, and Henry Chang. Intelligent file hoarding for mobile computers. In *First ACM International Conference on Mobile Computing and Networking - MobiCom'95*, November 1995. http://www.cs.columbia.edu/~lei/resume.html#publications.

[11] Douglas Terry, Marvin Theimer, Karin Petersen, Alan Demers, Mike Spreitzer, and Carl H. Hauser. Managing update conflict in bayou, a weakly connected replicated storage system. In *15th ACM Symposium on Operating Systems Principles*,
Copper Mountain Resort, Colorado, US, December 1995. http://www.parc.xerox.com/csl/projects/bayou/.