A File System for Mobile Computing

Carl Downing Tait

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

1993

ABSTRACT

A File System for Mobile Computing

Carl Downing Tait

Distributed file systems are a fundamental structure of distributed computing, and much attention has been focused on their design. But one important new design point has not yet been explored thoroughly: how to support mobile clients. Portable workstations are becoming increasingly common, and people who use these machines should not be forced to accept inferior performance or usability. This dissertation argues that two ideas — efficient variable-consistency replication and intelligent file prefetching — lead to a file system that supports mobile clients particularly well. Algorithms, implementations, and analyses are presented to support this assertion.

Contents

List of Figures

List of Tables

Acknowledgements

This dissertation is dedicated to my parents, whose seemingly unbounded love and support still amaze me, despite the fact that they've been that way for at least thirty years now.

Dan Duchamp, my advisor, has been of invaluable help — technical and otherwise — in getting me through the often arduous doctoral process. Dan also deserves thanks for enduring my numerous pianistic exploits, especially the 1990 National Chopin Competition.

The other members of my thesis committee — Anupam Bhide, Luis-Felipe Cabrera, Steve Nowick, and Sal Stolfo — are to be commended for their careful review of the dissertation and for their valuable comments.

The management of IBM Boca Raton provided generous financial support under the Resident Study Program during my first four years of graduate study.

Harry Harjono wrote a substantial portion of the user-level code for the file system prototype.

Bill Schilit's labor produced the file traces used in the trace-driven experiments described in this dissertation.

1

Introduction

Distributed file systems are a fundamental structure of distributed computing, and much attention has been focused on their design [3, 9, 13, 18, 29, 30, 31, 44, 51, 53]. But one important new design point has not yet been explored thoroughly: how to support mobile clients. Portable workstations are becoming increasingly common, and people who use these machines should not be forced to accept inferior performance or usability. This leaves an open question for research: what kind of file system is necessary or desirable to support mobile clients?

Portable workstations introduce new issues into file system design that have not yet been adequately addressed:

- The technology is currently different from non-portable machines: storage capacity is smaller, and network bandwidth is drastically reduced when wireless interconnection is used in a multi-access network. Wireless links currently have a speed of 9600 bits/second to 10 megabits/second, versus 10 to 622 megabits/second for typical hard-wired connections.

- Client mobility introduces transparency problems: for example, a client should be able to move around freely without having to provide reconfiguration information to the file system. Current systems require that clients play an active role when mobility is needed, and usually provide reduced performance even then.

- Temporary disconnection is likely. Although this is partly due to the current state of wireless technology, future users may well *choose* to disconnect temporarily to reduce the costs associated with using a wireless link. Therefore, a mobile client's cache must be managed with special care to maintain performance and availability. Disconnection and technology constraints make it particularly difficult to achieve high availability.

One might question the wisdom of choosing to run a *distributed* file system on a mobile computer. Why not just run a simple, local file system instead? There are, however, fundamental advantages in the distributed model:

- Read-sharing of files is possible, which greatly reduces the client's dependence on local storage devices.

- Writes made locally are automatically moved back to a "safe" server located elsewhere. In addition to safety, this allows write-sharing of files.

This dissertation describes two new ideas for effective support of mobile clients. These ideas will be briefly discussed in Sections 1.1 and 1.2. Later chapters will address the issues in depth. Throughout, our design is targeted for the "engineering/office" environment, in which many applications use many small files, but sharing is rare.[1] This is the standard environment for file system research.

1.1 Thesis

The central thesis of this dissertation is that two ideas — efficient variable-consistency replication and intelligent file prefetching — lead to a file system that supports mobile clients particularly well, under the constraints described above.

---

[1]We are using Ousterhout's taxonomy of file access patterns, which includes two other classes [38]:

1. Scientific jobs — large data sets read and written sequentially.
2. Transaction processing — frequent sharing, with strict consistency required.

1.2 Sub-Theses

This section briefly describes the two principal ideas at the core of this dissertation. Throughout, we use the "client-server" model: clients issue requests to servers in order to access or update files. *Replication* — storing copies of a file on more than one server — is used to improve file availability. If one particular server is down, a copy of the relevant file can usually be fetched from another server transparently.

1.2.1 Efficient Variable-Consistency Replication

Determining an equitable tradeoff between availability and consistency of data is a fundamental problem in file system design. A highly-available system may sometimes return inconsistent data due to server failures or network partitions; a highly-consistent system may be forced to deny service even when some replicas are available. The problems intensify in mobile computing. The presence of evanescent clients — connected one moment, disconnected the next, connected again sometime later (perhaps at a different location) — makes it difficult to achieve a high level of either consistency *or* availability, much less both.

Client mobility, combined with the likelihood of temporary disconnection, suggests that an unusually loose binding between clients and servers is desirable. Ideally, either the client or the server should be able to terminate the client-server relationship *at any time* with impunity. To reach this goal, however, we must take a new look at file system operations, and re-think the traditional ways of implementing them.

In current file systems, write operations are initiated by the client. Ordinary implementations require clients to block during writes because the operation must be synchronized with one or more servers. We propose instead a "lazy" or "write-back" scheme in which the client simply leaves updates in its cache. Servers periodically pick up these updates and propagate them to multiple replication sites.

The elimination of blocking during writes has several advantages for mobile operation. Client autonomy is increased: there is no need to synchronize write operations with servers; our system never blocks during writes unless a client's cache is full. This is particularly helpful for mobile clients, who may be distant from the servers handling their write requests. Furthermore, crash recovery is simplified because clients are no longer actively involved in propagating updates.

Reducing the interaction between clients and servers also improves *scalability* — the ability to support a large number of clients without significant performance degradation. This is especially important for mobile clients because server load will become more difficult to predict. As users move around, a server may suddenly find itself responsible for a large number of clients. We must be able to handle these transitory periods of high server load in a mobile environment.

In the worst case, our lazy approach to update propagation makes it more difficult to find what is provably the most recent version of a file. Furthermore, to insist on a uniformly high level of consistency would greatly reduce availability, and this conflicts with our desire to provide a highly-available file system. However, we hypothesize that most reads can be satisfied by returning any available copy; allowing the user to settle for this looser consistency can substantially improve performance and provide high availability. Therefore, we argue in favor of a small modification to the interface for read operations that will allow a less strict form of reads. The traditional read operation that is guaranteed to return the most up-to-date version of a file will still be supported, as well.

A client using a portable workstation should be able to move around freely without having to provide reconfiguration information to the file system, and without suffering severe performance degradation. Existing systems either perform poorly in these circumstances, or require the client to play an unduly active role in supporting mobility, or both.

We describe methods for supporting mobile clients in a way that is virtually invisible to the user, and that has a minimal impact on performance. In particular, we describe how to handle the difficult case of clients who are temporarily or permanently very distant from the "home site" where their files are replicated. The scheme we use is a logical extension of the approach described above.

1.2.2 Intelligent File Prefetching

For performance and, to a lesser extent, availability, it is highly desirable to prefetch files into a client's cache before the client actually accesses those files. Since we intend to support wireless portability, prefetching becomes even more important. Wireless links are relatively slow, so a cache that has been intelligently filled with prefetched files should markedly improve performance in this case. In addition, we desire a method that will be particularly effective in managing the relatively small cache of a portable workstation. (Skillful cache management is not as critical when a cache is large enough to hold most of the files that a client is actively using.)

We describe an intelligent prefetching algorithm that analyzes file access patterns in an attempt to predict which files a client will need in the near future. All of the work in deciding what to prefetch is done on the client side — putting this burden on servers would decrease scalability. As will be shown, both time and space requirements for the algorithm are modest.

Ideally, one might wish to be able to prefetch only portions of a file. It is beyond the scope of our algorithm to handle this case; however, it is an interesting area for future research, with preliminary work already done by Korner [25].

1.3 Outline of the Dissertation

Chapter 2 describes previous research in distributed file systems, providing a background for our own work. Chapter 3 describes the algorithms involved in efficient variable-consistency replication, including experimental data, comparison with existing systems, and description of a prototype implementation. In Chapter 4, the algorithms and implementation of intelligent file prefetching are discussed and analyzed. The work is summarized in Chapter 5.

2

Background

2.1 Introduction

At the heart of every operating system is its file system: the software that allows users to store and retrieve permanent data. In centralized operating systems, file systems are usually straightforward to implement. One simply keeps track of which disk blocks are free, which blocks belong to each file, and who may access a specified file. Some sort of directory structure is usually supported so that files may be organized hierarchically. File systems based on this approach are used by everything from MS-DOS to UNIX with unqualified success.

In a distributed operating system, however, the design and implementation of a good file system are singularly difficult problems. Even the description "good" is not well-defined. There are numerous trade-offs to consider, and predictably, designers have come up with a number of radically different distributed file systems.

In a centralized environment, the model is simple: a single computer with a number of storage devices attached to it. But a distributed system involves multiple machines communicating via some sort of network, with disk drives connected to some or all of these machines; a number of systems support diskless workstations.

Ideally, a user should not need to know where any particular file is stored. The file system should be able to locate a given file and make it available to a user. This feature, *transparency*, is the primary difference between "network" and "distributed" file systems. In a network system, the user is aware of the multiple machines in the environment, and is responsible for knowing which files are stored on which machines. At best, subtrees of files from other machines can be mounted locally, but machine-to-machine movement of files will still be visible. Multiple machines are available for use, which may well provide more power, but much of the responsibility for managing this power falls on the user. A distributed system relieves the user of a large part of this burden: file access is machine-transparent.

One natural technique for increasing the availability of files and improving performance in a distributed system is *replication* [4]: maintaining copies of each file on a number of machines. If one particular machine is down, a copy of the file can usually be retrieved from some other machine. And even if all replicas are available, performance can still be improved by keeping one or more replicas in close proximity to a client. Much of the time, however, a copy of the file will be both nearby and available, so the remaining copies will be largely unused. It is therefore unappealing for file system operations to expend a large amount of time or other resources in managing replicas of files. Mobile clients, with their less predictable appearances and disappearances, make the problem of efficient replica management even more difficult.

This chapter lays the groundwork for our own research by surveying and analyzing recent work in distributed file systems. Every file system must have some scheme for mapping character-string file names to the files they represent, so we begin with a discussion of naming. This is followed by a consideration of various replication schemes. The chapter ends with a discussion of caching, which can be used to improve both performance and availability.

2.2 Naming

File name resolution is an important part of any distributed file system. The central concern of a naming scheme is to establish a unique, distribution-transparent name for each file. When a client provides such a name, the system should be able to locate the file easily.

Name resolution needs to proceed swiftly, so the natural tendency is to make the table of name-to-file mappings widely available — at least as available as files — either through a name server or through replication. But both of these approaches have a devastating effect on scalability. A single name server is a performance bottleneck and a single failure point for the whole system, while a replicated name service introduces a new availability problem. In addition, for performance reasons, it should be possible for a file to migrate from one file server to another in a manner that is transparent to the user: server names should not be embedded in file names. Several proposed naming schemes are particularly noteworthy for how they address these issues.

Welch and Ousterhout [57] describe a name lookup mechanism known as *prefix tables.* This method is used in the Sprite system [35], which will be discussed further in Section 2.4. The distributed file system is seen as a single tree-structured hierarchy by users, but is actually divided into several *domains.* Each domain is a portion of the tree relegated to a particular server. A server may store more than one domain. Each client has a prefix table (typically incomplete) that maps file name prefixes to the servers on which the associated domains reside.

An example: suppose a domain on server C is rooted at `/chopin/etudes`. A client attempting to locate the file `/chopin/etudes/winter-wind` would find an entry for the prefix `/chopin/etudes` in its prefix table along with the information that this domain is on server C. Even if a domain with root `/chopin` is located on server A, the client will still know that its file is on server C, since the longest applicable prefix in the table is always used.

The prefix table method does not require that server names be included in file names. Furthermore, entries in a table are regarded merely as *hints* [55]. If a file is not where a table says it is, shorter prefixes are tried until the file is found. At each point, incorrect table entries are updated to reflect the new information. This gets around the update problem when a domain moves from one server to another: since table entries are just hints, they do not need to be absolutely correct at all times. File movement is therefore completely transparent to the user.

Finally, prefix tables can update themselves by exchanging information with other tables. If a client has no prefixes at all for a file (as will be the case initially), it broadcasts the file name to all servers. Relevant prefix/server mappings (with symbolic links already expanded) are sent back to the requesting client by all servers who have such mappings. In this way, prefix table information can be easily propagated around the network without the requirement that any two clients have precisely the same table. And because prefix tables contain only hints that need not be correct, this method avoids creating either an availability or a consistency problem.

A very different approach to naming is taken in the QuickSilver system [9, 19], which employs the concept of *user-centered naming*. Instead of all users having a single view of a global name space, each user has a logically distinct name space in which resolution is performed. It is possible to think of a global name space in which files are identified by (user, local filename) pairs, but conceptually, each user has an individual tree of files. Internally, QuickSilver associates a unique file identifier (UFID) with each file in the system.

The primary reason for this unusual scheme is a concern with scalability. In a global naming system, the lookup required to determine that a file does not exist may take time proportional to the size of the network, which does not bode well for scalability. However, any given user will have a relatively small name space that can be searched exhaustively, if necessary, without severe detriment.

Files are added to a user's name space through the use of file pointers called *links*, which may be either *soft*, *symbolic*, or *hard*. A soft link points to a UFID in another user's name space. A client establishing such a link is not charged for the space to store the file. However, the file may be deleted by the user who owns it, so there is no assurance that the soft link will be permanently accessible. Symbolic links are similar, except that the link is made to another user's local file name instead of to the UFID of that file. This is useful for ensuring that one is always reading the latest version of a file such as a compiler or text editor. Finally, a hard link creates a logical copy of a file, ensuring that it will be available until explicitly deleted by the user creating the link. A user making such a link is charged for the space that the file occupies.

An example will show how name resolution proceeds. Suppose user Rachmaninoff has established a soft link to Chopin's file `/preludes/c-minor` (UFID = 1234), and a symbolic link to Paganini's file `/caprices/number24`. Rachmaninoff's local names for these files are `/variations/chopin` and `/rhapsody/paganini`. When the links are established, file location hints are placed in Rachmaninoff's *user index*. The hints are created by employing a user-locating service called the *White Pages* to locate users Chopin and Paganini, and then searching their local name spaces for the desired files. Since binding of a symbolic name to a file must take place on every file access, the hint for the symbolic link is less specific than that for the soft link.

When Rachmaninoff attempts to access `/variations/chopin`, the hint is followed to reach the last known location of the file with UFID 1234. If this highly specific hint is wrong, QuickSilver next searches the disk index of the disk where file 1234 used to be located to see if it has moved somewhere else on the same disk. Next, other disks at that server site are examined. As a last resort, all server sites that store files belonging to Chopin — the owner of file 1234 — are searched. (The White Pages service provides a list of these sites if necessary.) In effect, QuickSilver works from specific hints to general ones, just as the prefix table method tries long prefixes before shorter ones.

One of QuickSilver's strong points is that a user can be physically relocated without a change in user-centered view. Forwarding information can be left behind in the White Pages to make the move transparent. Since this information is just a hint, it is discarded after some period of time. After the hint is discarded, other users' soft and symbolic links to the relocated user's files become invalid, and attempts to access files through these links will fail. New links must be established explicitly.

The Prospero file system [37], based on the Virtual System Model [36], uses a naming scheme that is similar in spirit to QuickSilver's. Prospero allows users to define customized views of a global file system. Clients can include or exclude any files they choose from their views.

A unique feature of Prospero is its support for the easy creation and maintenance of *derived views*: those that are not explicitly specified, but instead are derived from existing views. For example, a directory full of text files written in various languages could be used to create a derived view consisting of multiple subdirectories, each of which contains all the files written in one particular language. The *filter* used to create such a derived view must have access to attribute information necessary to distribute files within the new view. Once the derived view is created, however, it automatically reflects any changes that occur in the original view.

User-centered naming is an interesting concept, and one that is particularly useful when scalability and user relocation are concerned. Most systems, however, continue to follow the UNIX tradition and assume a uniform, hierarchical view for all users. Future systems will undoubtedly continue to use hints, which are unquestionably helpful in any naming scheme. Incorrect hints can simply be discarded, or if desired, updated.

It is certainly possible to imagine situations where resolution of a name with N hierarchical components would require N computers to be up: every component might require a different machine for resolution. So naming is tied to availability: if the name cannot be resolved due to unavailable information, the file cannot be accessed, even if it is on a machine that is up.

2.3 Replication

Replication is one of the most important issues involved in the design of a distributed file system. Without replication, high availability is unattainable: if a single server crashes, the file becomes unavailable. Maintaining multiple copies of a file is inherently costly, however. Not only is more disk space required, but complex software is needed to keep replicas consistent. In addition, network traffic will almost certainly become heavier.

One of the knottiest problems that arises in a replication scheme is that of detecting and handling inconsistent copies of a file. This is particularly likely to be necessary when servers are separated due to network partition. In that case, different replicas may be updated differently, resulting in divergent versions that may be irreconcilable when the network is reconnected. Directories are even more problematic. Losing file updates may be tolerable under some circumstances, but losing directory updates can cause the loss of entire files.

Conceptually, the simplest form of replication is that performed on a file-by-file basis, as is done in the Roe system [13]. Roe is designed to provide a single logical view of a heterogeneous local-area network. A *Roefile* is really a set of replicas, but the user sees only one logical entity.

In order to access a Roefile, the user issues a request to a *Transaction Coordinator*, which usually runs as a process on the user's machine. The coordinator treats the file access as an atomic transaction involving the set of replicas that the given Roefile comprises. The coordinator is also responsible for preserving enough information to recover in case of failure.

To translate the user's name for the Roefile into a set of file identifiers, the Transaction Coordinator makes use of the *Global Directory Subsystem*. This subsystem makes the required name translation, and obtains files from local file servers via each server's *Local Representative*. Because the network contains heterogeneous machines, these representatives are needed to maintain a uniform view of files for the directory subsystem across different local servers.

One of Roe's main goals is to enforce a high level of consistency among replicas. The *Weighted Voting* algorithm that the authors use meets their criteria nicely. Traditional quorum-based voting schemes using N replicas require that, if R copies are located for each read, at least N-R+1 copies must be written on each write. This ensures that each read will see an up-to-date copy of the file. If fewer than R copies are available at read time, the file is considered inaccessible.

In weighted voting, each replica has both a timestamp and a voting strength associated with it. Quorum size is based not on the number of replicas, but on the number of votes. Highly reliable servers can be given many votes, which usually reduces the number of copies that must be read or written to form a quorum. The same principle applies: if there are a total of V votes, and a read quorum consists of R votes, then the total voting strength of copies written must be at least V-R+1. Under this scheme, any quorum is guaranteed to contain an up-to-date replica of the given file. Furthermore, it is not necessary to worry about outdated copies, since only the replica with the most recent timestamp is read. When a file is written, all of its available replicas are given a new timestamp that is greater than the maximum of the old timestamps of those replicas.

Replicating directory information is somewhat harder. With individual files, Roe simply locks the file until the update is complete. But locking a directory for any period of time is clearly undesirable from the point of view of other users. Instead, Roe uses a callback scheme that requires users to *register* with directories that they want to access. When modifying a directory, a user attempts to update all replicas of that directory. If that is impossible, at least an appropriate write quorum must be gathered before writing, as with standard files. If some copies of the directory cannot be updated, all registered users are informed that they may no longer be using a current copy. Since a sufficient quorum was gathered before writing, however, it is easy for each user to obtain the most recent version of the directory when this occurs.

Another system that performs replication on a file-by-file basis is RNFS [31]. This system has high availability as a primary goal, unlike Roe, and can theoretically be implemented on top of any network file service. The authors chose Sun's NFS (Network File System) [24, 48] for several reasons, not the least of which was that it was readily available to them.

NFS has a stateless protocol: the server preserves no information between requests, so all relevant parameters must be included on each call. In addition, NFS read and write (but not control) operations are *idempotent*: calling a function arbitrarily many times with the same parameters has no more effect than calling it only once. RNFS clients can therefore recover from crashed servers simply by issuing their requests repeatedly until a response is received.

The high availability promised by RNFS is intended to be transparent to clients — the network file functions should not change visibly. To this end, RNFS interposes an *agent* process between a client and the actual file servers. Files are indeed replicated, but it is the agent's task to hide the replication details from the client.

The scheme used to ensure consistency is an extreme one that is optimal for reads: read one, write all (a quorum-based scheme with R = 1). When a server becomes unavailable, the agent makes a note of that fact in the *replicated file-list.* If a subsequent write is issued to a replica on a failed server, that copy is marked as invalid in the replicated file-list. When the server comes back up, it acquires exclusive access to the file, and replaces its bad replica with a valid one. If no writes were issued to a file while it was unavailable, no replacement is necessary.

Of course agents themselves may fail, and special care must be taken in this case. To begin with, the replicated file-list must itself be replicated on all servers in a form called the stable file-list. A recovering agent uses this file-list to verify that all supposedly valid copies of a file are identical, since an agent may have crashed in the middle of a write.

In order to prevent the entire system from being inaccessible during an agent failure, agents themselves are replicated. Clients may direct their requests to any agent they choose. If an agent goes down, the client just starts using a different one. A token-passing mechanism is used to ensure that two agents never attempt to write to the same file simultaneously: an agent must hold the token in order to write to the file.

Unfortunately, there is a substantial performance penalty caused by interposing agents between clients and servers, and by writing multiple copies of a file. The system's designers believe that they can fine-tune RNFS so that it is "no more than 1.5 to 2 times slower than NFS." Preliminary tests indicate that client caching will markedly improve performance.

After observing the performance penalty that RNFS pays for high availability through replication, it is not hard to see why commercially available systems such as NFS and AT&T's RFS (Remote File Sharing) [47] do not support replication at all. Their primary goal is to make remote file access convenient and (relatively) transparent. If the server for a desired file is down, the file is simply inaccessible. And if a file needs to be moved to a server that is closer to the client for performance reasons, the move will not be transparent to the user.

An unusual alternative to traditional replication is used in HA-NFS (Highly-

Available NFS) [6]. Unlike most systems, HA-NFS does not regard a processor and its attached storage devices as an indivisible unit with respect to failures. Instead, *dual-ported disks* are used so that a backup server may step in and *impersonate* a server that has failed.

During normal operation, a server keeps information concerning its volatile state (such as file system meta-data) in a log on disk. "Heartbeat" pulses are sent from the server to its backup at regular intervals, and if this steady flow is interrupted, the backup runs a failure-detection protocol. If the server has indeed failed, the backup server can take control of the failed server's disks thanks to the dual-ported mechanism. (Dual-ported disks are emulated by connecting normal disks via a bus.) The backup uses the failed server's disk log to reconstruct its state at the time it became unavailable.

Except for a brief delay before the backup takes over, and reduced performance due to the increased load on the backup (which is also acting as a primary server for other disks), this failure recovery mechanism is transparent to clients of the failed server. Such clients will continue to use the same server address and file handles as they did before the failure. In addition to transparency, HA-NFS has the advantage of very low overhead compared to standard replication schemes.

In the Andrew file system (AFS) [33], which will be discussed in detail in Section 2.4, replication is performed on groups of files known as *volumes* [52]. Because AFS replicas are read-only, replication is typically performed only on system files. A volume comprises an entire subtree of files, so replication of volumes may be wasteful. It is likely that a user will need high availability for only a few of the files in a volume, so replicating all of them is unnecessary. Aggregating files into larger units does tend to make the overall organization simpler, however.

The LOCUS file system [44] uses a compromise between volume and single-file replication. There is a single tree-structured name space for all files in the system. *Filegroups* can be mounted onto the tree in a manner analogous to mounting file systems in UNIX. Filegroups correspond to the AFS concept of volumes, but are replicated differently.

At every site where a given filegroup is to be replicated, a physical container called a *pack* is allocated for it. A pack can contain files from only one filegroup, but it need not contain all the files in the group. This allows individual files to have a high degree of replication without requiring that all files in the group be similarly replicated.

Furthermore, packs may vary in size, since a pack need only be large enough to hold the files replicated at that site. One pack is designated as the primary copy, and all members of a filegroup *must* be in that pack.

For each filegroup, one site is chosen as the *current synchronization site* (CSS). All requests to use a file in the group are directed to the CSS. Should the network become partitioned, a CSS will be created in each partition where the filegroup is used.

As its name implies, the CSS is responsible for ensuring harmonious interaction between clients attempting to use the same file. Tokens are used to synchronize reads and writes. There is also a *file offset token* that guarantees the correct offset within a file only to the client who holds the token.

A strong point of LOCUS is its ability to detect mutual inconsistency among replicas — even when some of the replicas have been renamed [42]. LOCUS uses *version vectors* to detect inconsistent replicas. A version vector has one component for each site where a replica is stored. When a file is written and closed, the corresponding version vectors at the sites where the file is written are incremented; update counters within the vectors serve as a kind of timestamp. Using version vectors along with the concept of a unique, immutable *origin point* for each file — when and where the file was created — LOCUS applies simple graph analysis techniques to determine if inconsistent copies of a file exist after a network partition. In complex situations, timestamping approaches may detect conflicts that do not actually exist, but the LOCUS method has been proven not to suffer from this drawback.

With respect to its replication techniques, the Ficus file system [18, 40] is the self-described "intellectual descendant" of LOCUS (both systems were developed at UCLA). Ficus is targeted for a very large-scale distributed computing environment, and most of its critical design decisions are centered on this problem.

The first and most central decision concerns the conditions under which reads and writes may take place. In a very large system, it is unreasonable to expect all servers to be up at all times (or realistically, at *any* time). Ficus therefore supports *one-copy availability*: under this policy (unlike quorum-based schemes), both reads and writes are allowed when as few as one copy of a file is available. The most up-to-date accessible copy is returned during a read operation; writes are initially performed on only one replica and propagated asynchronously to other replicas. This optimistic approach greatly increases availability, but allows the possibility of conflicting updates. Such conflicts, when they occur, are detected using version vectors as in LOCUS. Conflicting directory updates can usually be reconciled automatically, but conflicting updates to files require human intervention.

Ficus is implemented using the *stackable layers* model. At every layer in such a stack, the interfaces are identical: the interfaces a layer presents to its clients are the same ones it expects to find in the layer below it. Since all interfaces are uniform, layers can be inserted as needed to provide new services, and this is exactly how Ficus shoehorns in its support for replication. The Ficus *logical* layer is used to create the illusion that replication is not taking place; system calls using this layer see only one logical file. The logical layer sits on top of the NFS layer, which is used to access remote replicas. NFS calls pass through to the Ficus *physical* layer, which is used to manage individual replicas.

The interface between layers is a standard one: the *vnode* interface [24]. This "virtual inode" interface was chosen largely because of its familiarity and widespread use. This choice allows Ficus to use pre-existing UNIX and NFS services to handle the low-level details of local and remote file management.

Ficus uses *volumes* as a basis for replication, but these are actually closer in concept to the LOCUS pack than the AFS volume. Like packs, Ficus volumes act as containers for a designated set of files; they need not contain any particular file at a given time. Volumes are self-contained subtrees of files, and can be *grafted* into the Ficus name space much as one would mount a UNIX filesystem. During pathname resolution, volumes may be mounted automatically by the logical layer; this technique is known as *autografting*.

Ficus has been fine-tuned to support *primarily disconnected operation* [20]: a mode of operation in which the client and server are *usually* disconnected from one another. For example, a client may have one machine at home and another at work; ideally, these machines should be able to keep in sync with each other by connecting periodically and exchanging updates. A conflict detection method is necessary, but the version vector approach generates much network traffic and is quite slow in the primarily disconnected case. Therefore, simple timestamps are used instead: version vector comparison is necessary only for files that have been modified recently. In practice, the number of conflicts in a low-sharing environment has turned out to be quite small, due to what might be called a *human write token*: most updates to a file are made by the same person, who simply moves from machine to machine. Updates are thus implicitly serialized by the behavior of the human client.

Among replicated file systems, file-by-file replication is by far the most commonly used scheme. AFS groups files into volumes for ease of replica management, but this method requires all files within a volume to be replicated at each storage site. LOCUS's pack-based replication, shared in spirit by Ficus, is a happy compromise between the two extremes.

2.4 Caching

Caching is traditionally used to improve performance, but keeping an extra copy of a file in a cache can also increase availability. Perhaps the most significant decisions that must be made about caching are what to cache and where to cache it. This section will discuss two systems that answer these questions in completely different ways.

The primary goal of AFS, the Andrew file system [33, 50], is high scalability, and it is apparently successful in that regard [21]. In order to meet this goal, AFS takes great pains to reduce both server utilization and network traffic. When a user needs to access a file, the entire file is transferred to the user's local disk so that no further server interaction is needed until the file is closed.

Such whole-file caching is not an unreasonable approach. Indeed, several studies indicate that a high percentage of file accesses involve whole-file transfers

[5, 14, 39], so support for caching at a finer granularity would be wasted much of the time. Furthermore, whole-file caching is more likely to provide high availability, since entire files will be available in case of failure, instead of isolated disk blocks. Extremely large files such as databases, however, obviously cannot be manipulated in this way. The AFS designers are well aware of this restriction, but take the position that such support is provided by other means in their environment.

In the initial AFS implementation, a file in the local cache had to be validated before use. The client sent a message to the server requesting confirmation that the client's copy of the file was still up-to-date. Because files are usually updated in only one place at a time, this placed an unnecessarily heavy load on the server, so validation was subsequently abandoned and replaced with a *callback* scheme. A cached file is now assumed to be valid unless the system has explicitly invalidated it by sending a message to the client. AFS keeps track of which clients are caching a given file, and sends callback messages to all of them when a new version of the file is written. This modification has improved performance significantly, while continuing to ensure cache consistency.

The Sprite network file system [35], which is specifically designed for high performance, uses caching on both the client and server sides. Caching is block-oriented as in most centralized file systems. In addition, Sprite is intended to show the feasibility of diskless workstations, so all caching is done in memory instead of on disk as in AFS. If a file is concurrently write-shared, client caching is disabled so that a consistent view of the file can be maintained.

One unusual idea in Sprite cache management is dynamically varying the relative sizes of file cache memory and virtual memory. The Sprite designers have no objection to a cache occupying the majority of a user's memory, if not much space is needed for running processes. In fact, a Sprite file server uses the bulk of its memory as a file cache.

Block-oriented caching is more flexible than whole-file, but it is more expensive. It is a more complex model, and a harder one in which to maintain intra-file consistency. It is also likely to involve a heavier load on the server. On the other hand, whole-file caching appears to be an immutable design decision. It is not difficult to imagine a block-oriented system being modified to support whole-file caching as well, but it is very hard to picture the reverse.

The merits of various caching media are also debatable. A memory cache will clearly be faster than one on disk, but it will also be either smaller or more expensive (or both). In addition, if a crucial server is down for an extended period, there may be no way to save a file on a diskless Sprite workstation. (Because of Sprite's delayed writes, relatively brief server crashes may go unnoticed by the client.) Further, if a client crashes, data is more likely to be lost when using memory caching.

Either cache medium, however, will provide at least some amount of increased availability in addition to the improved performance once expects from caching. Even if a client is completely disconnected from the rest of the system, file data in the client's cache will still be available for use. Depending on the caching method used, this data may consist of anything from a single block of a file up to many separate files.

2.5 Summary

Table 2.1 summarizes the attributes and replication methods of various systems. Coda [51], the successor to AFS, has been tailored to support mobile computing, and will be discussed in detail along with our work in the following chapters. Note that both AFS and Coda use whole-file caching on a client's workstation disk, while Sprite uses client memory caching on diskless workstations. Sprite is scalable, but does not scale as well as AFS [21].

SYSTEM	REPLICATION UNIT	HIGHLY AVAILABLE	SCALABLE
AFS	Volume	No	Yes
Coda	Volume	Yes	Yes
LOCUS	Pack	Yes	No
Ficus	Volume	Yes	Yes
HA-NFS	—	Yes	No
QuickSilver	File	No	Yes
RNFS	File	Yes	No
Roe	File	Yes	No
Sprite	File	No	Yes

Table 2.1: Attributes of Existing Distributed File Systems

3

Efficient Variable-Consistency Replication

3.1 Introduction

This chapter investigates how to maintain replicas in a distributed file system, especially one supporting mobile clients. We argue that existing file replica management schemes would not cope well with mobile clients, and we present our solution: a lazy "server-based" update scheme and a new service interface that allows applications to select strong or weak consistency semantics on each particular read call.

The next two sections explain our design, both under normal operating circumstances and in the presence of failures. Section 3.4 consists of empirical data validating our ideas, and Section 3.5 describes a prototype implementation. Finally, Section 3.6 provides a qualitative comparison with existing systems.

We make several assumptions about operating conditions:

1. Client movements cannot be constrained, although patterns of movement may exist.

2. Latency of remote operations degrades as the distance between hosts increases.

3. The presented load is what Ousterhout has called "engineering/office applications" [38]. In this model workload, sequential sharing is not uncommon, but simultaneous sharing (other than read-read) is rare.

4. A file cache of modest size is maintained by each client.

5. File service sites can synchronize their clocks. This assumption, once controversial, can now be satisfied by several clock synchronization protocols (e.g., NTP [32][1]). We do not make any assumptions about client clocks.

Flowing from these assumptions is our conclusion that three design goals are of paramount importance in handling mobile clients: we must minimize synchronous multi-server operations, ease the addition and deletion of server sites, and allow for incomplete replication at servers. We discuss our reasoning for each in turn.

Minimize synchronous multi-server operations. Our first two assumptions lead us to conclude that file systems that frequently use "global communication"[2] will not perform well if clients move over a wide area while they remain mapped to a fixed set of servers. The reason is that the latency of contacting the most distant server is a lower bound on the latency of the entire multi-server operation. File service operations that involve global communication will slow down when a client moves away from its current set of servers.

Ease the addition and deletion of server sites. In order to avoid global communication, two features seem desirable: first, a primary-secondary server organization, in which the client communicates only with the primary and the primary communicates asynchronously with the other servers; second, ensuring that the primary is always located "near" the client (perhaps as measured in network hops). Since the client moves unpredictably, the service should be ready to regularly add a new replication site, *typically as the new primary.* The model of mobile operation we envision is that a client moves over a wide area, perhaps to areas it has never visited

[1]NTP consists of a self-organizing hierarchy of primary and secondary time servers. *Synchronization distance* — the round-trip message delay involved in communicating with a primary time server — is taken into account to synchronize clocks.

[2]We define a global communication as an operation in which several servers must be contacted synchronously. Coda's `close()` operation is an example in this category.

before. Once in a new milieu, it will negotiate with some local machine to become its new primary. There is also presumably a need for regular replica deletion, in order to limit the degree of replication to a sensible number in case of a highly mobile client. Regular addition and deletion of replicas, especially the primary, marks our design as unusual.

Allow for incomplete replication. If identities of the servers — specifically, the primary — are changing frequently and unpredictably, replication sites should not be expected to always store complete copies of the client's file set. We provide for incomplete replication, in which a client's files need be replicated at only a subset of the server sites.

3.2 Operation in the Absence of Failures

As suggested in the introduction, our design reduces global communication by using a primary-secondary server hierarchy. Typically, the client need not contact *any* servers, but even in the worst case, the client communicates synchronously with the primary only. We employ write-back caching with the primary, not the client, choosing when updates are copied from the client's cache. The primary makes periodic *pickups* from the clients it is servicing, and propagates updates back to the secondaries asynchronously. This pickup strategy allows the client's `write()` operation to return immediately after placing the new value in its cache.

Once some number, N, of secondaries have acknowledged receipt of an update — and the primary has verified that the update is *currently* present on at least N secondaries — the primary informs the client that the associated cached update can be discarded. This notification — which might be piggybacked onto the next pickup request — is called a *purge notice*. Because a client must retain an update in its cache until receiving a purge notice, the service has some latitude (constrained by the size of the client's cache) to "wait out" primary and secondary server failures without any disruption in accepting `write()` calls.

(For reasons that will be discussed in Section 3.3.5, the number, N, of replicas that must be written before an update can be discarded must be at least a majority of secondaries. This restriction is necessary to detect conflicts properly.)

During periods of heavy update activity, a client's cache may fill between primary server pickups. For this case, we provide a synchronous, blocking *forced write* operation that causes an immediate pickup.

The pickup interval can be dynamically modified during normal operation, based on the client's update activity. (Experimental results in Section 3.4.2 suggest bounds on the pickup interval.) The longer the pickup interval, the more short-lived files and associated update operations are never seen by the servers. Previous work on file caching [5, 21, 35] has shown that having clients lazily write back updates substantially improves scalability.

Asynchronous update propagation has well-known benefits and dangers, and it has been used in the design of other replicated file systems (e.g., [53]). Our work is significantly different in two ways. First, by requiring that the client retain an update until a sufficient number of secondaries have the new value, we trade cache space for a low-latency, yet reliable write operation. Other systems typically block while some number of replicas are written (the remainder being written asynchronously), or return immediately with no guarantee that the update will be safely propagated. We push back the blocking point so that the client typically is not involved in the propagation process.

The second feature that sets our work apart is how we couple asynchronous update propagation with a read interface that allows an application to choose either "lazy" or "UNIX-like" semantics on a per-read basis, as described below. When there is little sequential sharing, this design achieves high fault tolerance and low latency for both reads and writes. Experimental data will be presented in Section 3.4 to support the feasibility and potential value of this *dual-read-call* interface.

Given such a design, the key issues are:

- How does this interface work?
- When, why, and how do clients and primaries bind to each other?
- How are files replicated on secondaries?

We address these issues in the following subsections.

3.2.1 Dual-Read-Call Interface

Lazy operation is unsuitable for applications that share a file simultaneously or that perform write-read sharing with the read closely following the write. Such applications will typically want to see each other's updates as soon as possible. In our design, an application program can choose between weak or strong semantics on a per-use basis.

We abandon the traditional `read()` interface in favor of two alternatives: `strict_read()` and `loose_read()`. There is no guarantee concerning the value returned by the loose form, whose implementation is shown in Figure 3.1. In principle, the strict form returns the "most recent consistent" value; this will be defined more precisely after we have explained the operation of `strict_read()`. If `strict_read()` and `write()` are used exclusively, the system provides "one-copy UNIX semantics" (1USR): the semantics found in a centralized UNIX system. 1USR is not equivalent to one-copy serializability (1SR): due to caching, 1USR clients can make conflicting updates to a file without knowing that they are doing so. For compatibility, a generic `read()` library call could simply call `strict_read()`. For the convenience of some applications, another library routine could be defined with the following semantics: strictly read the requested file if possible; otherwise, return any available copy.

While `loose_read()` is allowed to return any convenient value, analysis of file-trace data that will be presented in Section 3.4 indicates that a "best effort" implementation will almost always return the most up-to-date value. This conclusion is supported by the recent study by Baker et al. [5], which found that only one third of one percent of `open()` calls read data that was written by another client less than 60 seconds previously. This study concluded that (automatic) cache consistency is desirable to prevent clients from using stale data; in our system, clients use the `strict_read()` operation to enforce cache consistency.

Division of `read()` into `loose_read()` and `strict_read()` ensures that the

```
Look in the client's cache

If there is no copy in the cache, then check the primary server

If there is no copy at the primary,
   then check any of the secondaries, in any order
```

Figure 3.1: Loose Read Algorithm

entire cost of establishing 1USR consistency is charged to the process (reader) that demands the consistent value. This approach is in contrast to the other systems we know of, which take the "immediate write-back" (IWB) approach to consistency and availability. That is, when a file is either written or closed (depending on the system), the new value is copied to one or more servers before the client's synchronous call returns.

The eager approach of IWB seeks to support sharing. Unfortunately, IWB incurs the cost of synchronous global communication on every write/close regardless of when the next read happens. Since in most cases of sequential sharing the read occurs quite some time after the write,[3] the IWB approach imposes extra overhead on a workload that includes little sharing. This approach becomes even more unappealing in the presence of mobile clients, who will leave "trails" of updates as they move around. Searching these trails is time-consuming, and we wish to avoid such searches whenever possible.

[3]See Section 3.4 for empirical support for this contention.

3.2.2 Currency Tokens

A naive implementation of `strict_read()` would contact all servers and all clients that had read the file and retrieve the most up-to-date copy it finds. *Currency tokens* (CTs) are used to avoid this naive approach.

CTs rely on the idea of a *potential consistent writer*, or PCW. A PCW is a client site with a process that has strictly read a file, and that has write permission for that file. In other words, a client is declared a PCW after demonstrating both desire (strict read) and ability (write permission) to make an update in a consistent fashion. A CT is given in response to a `strict_read()` if there are no PCWs for that file, or if the client is the only PCW. A client can receive a CT only from a strict read, never from a loose read. Holding a CT allows the client to know that either there are no PCWs for the file in question, or that all potential updates are localized to itself and its primary-secondary hierarchy.

A client that performs a strict read without a CT initiates a relatively complex series of actions. First, recall that an update must be replicated on at least N secondaries before a client is allowed to purge the update from its cache. Therefore, assuming that there are a total of T secondaries, at least T-N+1 secondaries must be contacted — as well as any PCWs that have the file in their caches — to ensure that the most recent value is located. This condition is simply that of quorum consensus, wherein the read and write quorums must overlap [16].

Files are tagged with timestamps so that the copies at different replicas can be compared for recency. After reading T-N+1 copies, the primary compares the timestamps and gives the client the most recent copy of the file plus an indication of whether it has a CT. To ensure consistency, lists of PCWs and clients who hold CTs must be maintained in non-volatile storage on at least a majority of secondaries so that this information will be found during any initial `strict_read()`. The complete algorithm is sketched in Figure 3.2.

If a currency token cannot be gained, a client may still perform strict reads by executing the above sequence of actions on every read operation. In the typical case, however, the client *will* receive a CT after the initial strict read, and this makes

```
Phase 1:
   Primary multicasts strict_read() to all secondaries

   Each secondary evaluates whether client is a PCW, and,
      if so, records the fact in non-volatile storage

   Each secondary returns a timestamped file to the primary,
      along with timestamped lists of PCWs and CTs for this file

Phase 2:
   Block until at least a majority of secondaries reply

   Using the most up-to-date PCW list it has received,
      the primary determines that client should be given a CT
      iff there are no PCWs, or client is the only PCW

   If a new CT is being granted, this fact is recorded in
      non-volatile storage on at least a majority of secondaries

   If old CTs must be revoked due to the arrival of a PCW,
      the primary performs the revocation

Phase 3:
   Primary reads cached copies from all PCWs

   Primary returns most up-to-date replica gathered from
      secondaries and PCWs, along with a CT if appropriate
```

Figure 3.2: Strict Read Without a CT

```
Look in the client's cache

If there is no copy in the cache, then check the primary server

If there is no copy at the primary, then check at least T-N+1
   secondaries and return the most recent value found
```

Figure 3.3: Strict Read With a CT

subsequent strict reads considerably simpler and faster. The algorithm for this case is shown in Figure 3.3; note the similarity to the loose read algorithm in Figure 3.1. CTs have the effect of allowing most strict reads to be implemented by executing almost the same sequence of actions that is performed for a loose read. The only difference is at the level of secondaries: if the search gets that far, a strict reader with a CT must contact enough secondaries to ensure that the most recent copy is located. The initial strict read is the only point in our design at which a synchronous operation is required: several secondaries are contacted, as well as all PCWs.

It might appear advisable to tie the initial strict read to the `open()` call, but in practice, it makes little difference. The complicated initial strict read algorithm must be performed before the first byte of the file is returned; whether this code is executed in conjunction with the `open()` or the first `strict_read()` is usually irrelevant. For greater flexibility, however, we have decided to make `open()` as simple as possible. This allows a client to choose between strict and loose read on every read call; the strict/loose decision need not be made at the time the file is opened.

PCW and CT lists are updated at secondaries by a simple process that can do only one update at a time. It is necessary to wait in line to have a PCW or CT added to (or deleted from) the list. Furthermore, since we allow initial strict reads

OPERATION	QUORUM REQUIREMENT
Write	Asynchronously write file at N secondaries, where N is at least a majority
Initial strict read	Read file at T-N+1 secondaries Read/write PCW and CT lists at majority of secondaries
Strict read with CT	Read file at T-N+1 secondaries if secondaries read at all

Table 3.1: Quorum Conditions

from different clients to execute simultaneously, a majority of secondaries must be contacted to ensure that at least one client learns about the other.

It would therefore seem necessary to contact a read quorum of secondaries (T-N+1) or a majority, whichever is larger. However, because a write quorum (N) must be at *least* a majority, a read quorum is at *most* a majority; hence, contacting a majority of secondaries during the initial strict read is guaranteed to fulfill the read quorum requirement. Later strict reads with a CT need contact only a read quorum, not a majority (if secondaries need to be read at all). A complete statement of the quorum conditions is given in Table 3.1.

Correctness of Simultaneous Operations. There is one more wrinkle necessary to ensure that simultaneous initial strict reads behave correctly. After updating at least a majority of CT lists when a new CT is granted, the primary must validate that no new PCWs arrived while the CT lists were being modified. This requires that a majority of secondaries be contacted again to retrieve the latest PCW list. If a new PCW *has* arrived, the new CT is now invalid, and must be deleted from at least a majority of CT lists.

This validation mechanism, combined with the fixed sequence of operations involved in gaining a CT, implicitly serialize initial strict reads. Overlapping strict reads of the same file are therefore assured of functional correctness. To see why this is true, assume that two clients start an initial strict read at the same time:

1. Two non-PCWs: No problem, of course.

2. Two PCWs: Because both clients are required to validate their CTs (if one was granted), each client will learn of the other's existence, and belatedly revoke its own CT.

3. A non-PCW and a PCW: The same reasoning as (2), although only the non-PCW will need to revoke any tentatively-granted CT.

A small complication arises because we allow *incomplete replication*; that is, there is no requirement that a file be replicated at every secondary. It is possible that some secondaries will not have the file at all, much less the most recent version. Nonetheless, by using the scheme described above, we ensure that an initial strict read will find the most recent replica, assuming that a sufficient number of secondaries are available. A secondary that does not have a replica of the file in question acts as a kind of *witness* [41]: a server that may be counted as part of a quorum even though it contains no explicit information about a file.

Multi-File Currency Tokens

A file that has several names — through symbolic links, for example — is covered by a single CT; it is not necessary to gain a CT for each name. In general, however, we have rejected the seemingly attractive notion of allowing a CT to cover multiple files (say, all files within a directory) rather than a single file. The problem is that gaining and using a multi-file token is typically no faster than gaining a CT for each file individually. Furthermore, after spending a non-trivial amount of time in expanding the token, a single additional PCW can topple the carefully constructed CT edifice.

The unavoidable difficulty in a multi-file CT scheme is that the most recent version of each file covered by the token must still be sought out and copied into the local cache when the file is strictly read for the first time. The CT tells us that this client is the only PCW (or there are no PCWs), but this does not relieve us of our obligation to gather a read quorum for each strictly-read file.

An example: suppose a client strictly reads `/dir/fileA` and gets a multi-file CT covering all of `/dir`. Because the service has not explicitly searched out the most recent versions of any files in `/dir` other than `/dir/fileA`, the service must still contact a read quorum of secondaries when the client strictly reads `/dir/fileB`. This is true of every file the client strictly reads in `/dir`, so the system might as well acquire the CTs one at a time.

Currency Token Revocation

CTs can be revoked for several reasons. The guiding principle is that the presence of even one PCW for a file requires all other clients to give up their CTs on that file. For example, if one or more clients have strictly read a file to which they do not have write access, and a PCW for that file arrives, existing tokens are revoked, and the unique PCW is given the only CT. If a second PCW then strictly reads the same file, all tokens are revoked; no clients are allowed to hold CTs for a file if there are two or more PCWs.

What happens if the service needs to contact a currently unreachable client, either to revoke a CT or to read a file from a PCW's cache? A straightforward solution involves timeouts: a client who cannot be contacted for a certain length of time is deemed to have become permanently disconnected. CT and PCW information concerning the client is discarded, and other users' strict reads are allowed to proceed. But this approach cannot completely close the window of error: a client who has been disconnected might suddenly reappear with a cache full of updates that were missed by strict readers in the interim.

Although we cannot eliminate errors entirely, we can at least eliminate the inelegance and delay of timeouts; we have concluded that demand-based disconnection is a convenient simplification. Whenever the service needs to contact a client, but the client is unreachable for some reason, we give up immediately, automatically revoking any CTs that the wayward client is holding. This applies with equal force whether the system is making server-based pickups, attempting to contact PCWs, or explicitly revoking CTs.

It is important to realize that CTs are not equivalent to read and write tokens used by other systems. They are simply *hints* [55] used to improve the performance of strict reads. In other systems, tokens are prerequisites for performing operations; in our design, strict reads can take place without using currency tokens, though performance will be less than optimal. CTs are somewhat akin to *callbacks* in AFS [33] and Coda [51], though the pessimistic nature of CTs provides a stronger consistency guarantee: a CT will not be granted if any other client has even the *potential* to make a consistent update.

Revocation of PCW Status

As described so far, a client who becomes a PCW for a file remains a PCW indefinitely. This poses problems for other clients who might wish (for example) to strictly read a program that a PCW has made available for public use. The existence of the PCW means that no CTs can be granted to other clients. This requires them to contact the PCW on every `strict_read()` — even if the program they are reading has not been modified in months or years.

A simple, ideal fix for this problem is to have the PCW explicitly mark the file as read-only, thus taking itself off the PCW list. For PCWs who do not bother to do this, we provide an alternative solution. Periodically, a trusted program is run that shuts out all client requests and analyzes the PCW lists to see which clients should still be considered "potential writers" for a file. If the file in question has not been modified for a suitably lengthy period, the PCW loses both its PCW status and its CT, if it has one.

It would be better if PCW revocation could be allowed to take place during normal operations. Because clients are not trusted, however, we cannot prevent a client from initiating a new strict read at any time — even while that client's PCW status is being revoked. A global lock on PCW lists would allow on-demand PCW revocation, but this is a poor solution. Aside from the inherent undesirability of global locks in a distributed system, our design is lock-free, and we see no reason to introduce an ugly global lock here. PCW revocation is simply a performance enhancement; from the standpoint of technical correctness, a client can remain a PCW indefinitely. We are therefore willing to use a relatively inelegant scheme —

executed only periodically — for the purpose of revoking PCW status.

Similarly, CTs can persist for very long periods. In order to keep CT lists from growing indefinitely, it is advisable to periodically check the lists to ensure that the associated clients are still reachable and are still interested in holding those tokens.

3.2.3 Further Details

Interference Between Strict and Loose Reads. It is the responsibility of clients to use strict reads when a file is expected to be shared. However, a server can optionally retain memory of all clients that have read files from it, whether these reads were performed strictly or loosely. In this case, the service can detect potential interference between `loose_read()` and `strict_read()`.

It is perhaps worth mentioning two cases in which the same file might reasonably be read both strictly *and* loosely by the same client:

- Loose, then strict: A client browses through a number of files loosely, then strictly reads one of them in order to make a consistent update.

- Strict, then loose: A client strictly reads a file, but does not receive a CT. The client then becomes disconnected and decides to continue reading the cached copy using loose read.

Optimization for Write-Read Sharing. We provide a "short-circuit" optimization for the single-writer, multi-reader case. If there is only one pre-existing PCW at the time of a `strict_read()` operation, and the reader is not itself a PCW, then the reader can request the file from the unique PCW. If the PCW still holds a CT on that file, and the file is in its cache, it delivers its value to the reader. Thereafter, strict read can be performed by direct client-to-client communication, reverting to the usual method only when the PCW loses its CT.

Write-Write Sharing and File Locking. Unlike systems such as Deceit [53] and Echo [30], we do not use the concept of a "write token" in our design. This means that two PCWs may simultaneously modify their cached copies of a file, and this will be detected as a conflict during update propagation. Although we have not provided explicit support for file locking, it would be possible to add a distributed lock mechanism (such as that used in [45]) on top of our design. Such locks would allow clients to gain exclusive access to a file, preventing possible conflicts. However, we have taken care to avoid the use of *mandatory* global locks in our design.

Create and Delete Operations. Programs such as compilers and text formatters often create intermediate files as they run. Since these temporary files are usually short-lived, there is no reason to write them to servers. Therefore, newly-created files at the client are not accessible to other clients by any form of read until they are propagated to servers — if they live long enough to be picked up. Newly-created directories are handled in the same way.

Deletion of a file or directory is trickier. Replicas at secondaries are deleted immediately, but copies held in clients' caches cannot be dispensed with so easily, for two reasons. First, we must ensure that we do not delete a client's file while it is still open. We get around this problem by adding an entry to the client's *deletion log*, which lists the files and directories that are to be deleted when the client determines that they have been closed.

Second, the service is not guaranteed to know the identities of *all* clients who have read a file into their caches — only the PCWs. So log-based deletion is performed only at PCWs, which is acceptable: clients who did not strictly read the file in question receive no guarantees as to what will happen if they attempt to update it. The file continues to exist in loose readers' caches until bumped out by subsequent file operations.

Control Operations. Because of the need to keep track of PCWs, control operations that change the protection attributes of a file must be done in the manner of an initial strict read: a read quorum must be gathered, and a write quorum must be written. This approach suffices for technical correctness, but due to the high frequency of `stat()` operations, it is preferable to write to all replicas when changing a file's attributes. `stat()` can then be performed on any replica with a high degree of confidence that the correct attributes will be returned. `rename()` operations should also be performed on all available replicas; the new name will be usable on subsequent `open()` calls.

Hard and symbolic links create new names for files, and are thus handled in the same way as file creation. Link information is written only locally, and is not accessible to other clients until propagated to secondaries during the next pickup operation.

Directories themselves are handled in a somewhat unusual way. Clients do not make explicit directory updates in the same way they make file updates: directories are essentially *local* constructs that are used to store metadata about the files at any particular client or server. The file service is responsible for propagating modified metadata on a per-item basis during control operations.

3.2.4 Client-Primary Attachment

Client-to-primary assignments are made by a special module known as the *matchmaker*, which is most conveniently implemented as a process on any service machine. When a client wishes to obtain a primary server, it sends a request to any convenient matchmaker, which then selects a primary based on criteria of its choice. For coping with mobile clients, the criterion might be physical proximity.

After making its choice, the matchmaker sends the client's address to the selected primary and forgets about the transaction. The new primary server then performs a pickup so that the client will learn the identity of its new primary. The client saves the primary's address in (volatile) storage for use in subsequent read operations.

The only non-volatile state retained by the matchmaker is a list of servers that can act as primaries. It is not necessary to modify this list when a potential primary crashes. The matchmaker can "ping" the machine it has chosen for a given client, and if there is no response, the matchmaker will simply choose a different server. Modifying the list of primaries is trivial. Adding a new server poses no problems, and deletion is not difficult. If a client attempts to use a primary that has been deleted from the list, either there will be no response, or the message will be rejected by the disgruntled ex-primary. In either case, the client need only apply to the matchmaker to obtain a new primary.

Given the relative infrequency of calls to the matchmaker, we do not anticipate any scalability problems. For availability, however, it is desirable to run the matchmaker on several different machines. Since the matchmaker is stateless except for the list of potential primaries, multiple executions on different machines can operate independently if necessary.

A client may ask the matchmaker for a new primary at any time, for any reason. Because the client does not discard cached updates until receiving confirmation that they have been stored on the required number of secondaries, switching among primaries in any arbitrary way is no threat to correctness. Similarly, a primary may safely pull out of a client-primary relationship at any time. In both cases, the client will have to obtain a new primary and pickups will resume normally.

Good reasons for requesting a new primary include failure of the old primary and movement (leading to reduced performance) by the client. Letting the client drive the selection process substantially simplifies our design, especially in the face of failures (see Section 3.3.2 for details). There is a cost for changing primaries: the matchmaking protocol must be run, CTs must be revalidated, and updates must be picked up once again by the new primary, so the client should not be capricious about requesting a new primary.

When a switch is made, both the new primary and the client will learn about it: the new primary's first job is to inform the client of its identity. In order to handle primary changeover gracefully without resorting to explicit disconnection messages, old primaries *figure out* that their services are no longer needed. This is straightforward because the client always knows the identity of its current primary. Whenever a putative primary goes to a client to make a pickup, it first verifies that it is still the primary server for that client. If the client is now being serviced by a new primary, the old one goes away and leaves the client alone. For example, suppose that client C has moved away from primary P1, and has called the matchmaker to obtain a new primary P2. During P1's next pickup, it learns that P2 is the new primary. P1 immediately stops making pickups from C.

It is tempting to blur the line between clients and primaries: why not run the primary server code on the client machine? This would reduce the number of messages required during an initial `strict_read()`. Unfortunately, clients are not trusted, so we cannot allow them to access secondaries directly. A malicious client could invalidate our consistency guarantees by ignoring our protocols and writing inconsistent values to secondaries.

3.2.5 Filesystem-Secondary Attachment

Primary servers are used principally as intermediaries between clients and secondaries. Primaries do not function as replication sites *per se*. Instead, we use groups of secondary servers to replicate file systems. For every file system, there is a corresponding group of secondary servers that handles the replication of files in that file system. Because we support incomplete replication, it is not necessary that every file be replicated on every server, however.

In practice, a server can simultaneously act as a primary and a secondary. A group of machines may be used both to replicate files and to act as primary servers for clients. Logically, however, it is necessary to maintain the functional distinction between primaries and secondaries.

A primary server learns about filesystem-to-secondary mappings by calling a *mapping server* specified by the client. Ordinarily, this `map()` function will be called only when a file system is mounted. This is a comparatively infrequent operation, so scalability should not be adversely affected.

We assume that secondary server mappings will rarely be changed, but we do provide a modification method based on Gray's idea of *leases* [17]. Mappings returned by `map()` are valid for only a limited period of time: relatively long (say, 10 to 30 minutes), but decidedly non-infinite. At the end of the lease period, `map()` must be called again to revalidate the mapping.

To modify a filesystem-to-secondary mapping, we must wait until all leases

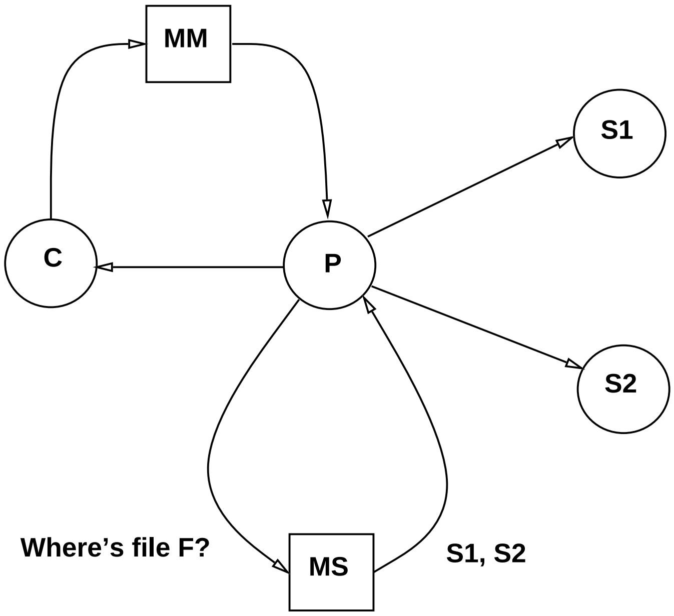

Figure 3.4: System Organization

on the mapping expire. At that point, we can update the mapping and let `map()` start handing out the new information. In order to prevent starvation while waiting for leases to expire, the mapping server can hand out shorter leases after there has been a request to re-map. For example, if we know that all leases will expire in at most 30 minutes, we can still provide 29-minute leases on the same information.

The overall organization of our system is shown in Figure 3.4. There, C is the client, P is the primary, S1 and S2 are secondaries, MM is the matchmaker, and MS is the mapping server. Table 3.2 shows the state information that must be held by each component in the system, and whether the state needs to be held in non-volatile or volatile storage. In that table, the notation "File+timestamp cache?" refers to the fact that the client may store its updates (and their associated timestamp information) in either volatile or non-volatile storage; clients that store updates in non-volatile storage will not lose data in a crash.

COMPONENT	NON-VOLATILE STORAGE	VOLATILE STORAGE
Client	Own address Mapping server address File+timestamp cache?	Current primary CTs held File+timestamp cache?
Primary		List of clients Filesystem-to-S mappings Files being propagated
Secondary	List of PCWs List of CTs	
Matchmaker	List of primaries	
Mapping server	Filesystem-to-S mappings	

Table 3.2: State Held by Each Component

3.2.6 Connection at a Distant Site

An intended feature of our design is that a client can move to any location where network access is available and still continue to access the files at its "home site" in the same way that it would access them locally. For example, someone from New York who is visiting Berkeley would call the Berkeley matchmaker to obtain a local primary server. (We assume that some form of negotiation is made for the use of local resources.) The locally-chosen primary then calls the mapping server in New York to locate files that the client wishes to access. The client need remember only its own address and the address of the mapping server at its home site. These addresses need to be retained in non-volatile storage. Previous work on mobile internetworking by Ioannidis et al. [22] has demonstrated the feasibility of such a scheme from a networking perspective.

Because our system allows a client to use exactly the same name for a file regardless of the client's physical location, we have effectively made naming an issue that is orthogonal to the rest of our design. A client can use any naming scheme that is understood by the mapping server at the home site; this scheme persists across all client movement.

During operation at the home site, we do not expect the primary server to play an important role as an intermediate cache. Muntz and Honeyman [34] report that intermediate caches experience a surprisingly low hit rate: less than 19% when the client itself has a 20 megabyte cache. Blaze and Alonso [7] achieve better results by using a dynamic, hierarchical caching scheme. In their model, however, clients are assumed to have minimally-shared hierarchies such as home directories and temporary files stored on local disks, which is not an assumption we wish to make in our system. When the primary is far from the secondaries, however, even a relatively low intermediate cache hit rate is a significant advantage for a client whose files would otherwise have to be shipped from a distant site. Furthermore, our design is not limited to disk-based caches; a memory-based cache of only a few megabytes would experience a higher hit rate at the primary server.

3.2.7 Mobility at a Distant Site

Since a newly-selected primary always starts with an empty cache, a client who switches primaries while connected at a distant site faces the problem of having to fill the new primary's cache from scratch. To mitigate this problem, the client can remember the address of its old primary, and request that the new primary lazily copy files from the old cache. The old primary has no obligations in the matter; as soon as the client moves to the new primary, the old cache is subject to re-use by other clients. As long as the cached files have not been overwritten by other clients, however, this use of the old primary can reduce the number of cache misses experienced by the client at the new primary. In effect, we allow files to "follow a client around" during movement at a distant site.

3.3 Failure Recovery

A major advantage of our design is that it makes the algorithms for failure recovery extremely simple. In this section, we describe how crashes and partitions are dealt with.

3.3.1 Secondary Server Failure

Failure of a secondary server will not be noticed unless so many secondaries fail that an initial `strict_read()` can no longer guarantee consistency. It is also possible that `write()` operations will block: if the primary is unable to propagate updates to a sufficient number of secondaries due to failures, purge notices will not be sent, and the client's cache may fill to capacity with pending updates.

Recovery is easy: the recovering secondary is not required to do anything when it comes back up because an initial strict read will always contact enough secondaries to ensure that the most recent value is located. However, finding and copying over the most recent versions of files is a good idea because it will increase the likelihood that a `loose_read()` directed to the recovered server will return up-to-date values. The process of getting up to date can be done simultaneously with accepting updates — there need not be a synchronous "recovery phase" as is required, for example, in Echo [30]. This is a consequence of the quorum requirement: there are no assumptions made about which secondaries have the most recent values.

3.3.2 Primary Server Failure

It is the client's responsibility to detect when its primary server has crashed. When its requests go unanswered, the client asks the matchmaker for a new primary. By letting the client drive the process, and by recognizing that the new primary is not required to have up-to-date versions of any files, we avoid the complexity of a traditional election scheme [15].

Furthermore, a primary can crash (or just drop out) at any time without causing updates to be lost. Because a client is required to hold an update in its cache until receiving a purge notice — at which point the update is replicated on N or more secondaries — it is impossible for a primary to hold the only copy of an update. The newly-chosen primary will automatically restart the update propagation process for any updates remaining in the client's cache. No special logic is required; the new primary treats this case in exactly the same way as regular update propagation. The only minor inconvenience is that a client's currency tokens must be revalidated immediately after attaching to a new primary; the client-primary-secondary hierarchy upon which CTs rely may have been temporarily disrupted by the crash. Revalidation is quite simple: the primary gathers up-to-date CT information on the client from the secondaries, and gives this validated list of CTs to the client.

Recovery is trivial, again because all the state the primary held can be regenerated elsewhere. A primary that has come back up simply waits for the matchmaker to pair it up with clients. The recovering primary does not retain any state pertaining to its interactions with previous clients.

3.3.3 Client Failure

As in any write-back caching system, it is possible to lose data held at the client when the client crashes. Due to the use of asynchronous update propagation, our window of danger is somewhat wider than in other systems. However, if the client's cache is non-volatile, updates need not be lost, even if they are unavailable when a client is disconnected.

When a primary server is unable to make a pickup from a client, it assumes that the client has crashed and so it stops making pickups. If the client has not crashed, no harm is done: the client will eventually ask the matchmaker for a new primary. This is exactly what the client will do during recovery if it actually has crashed. Note that this scheme works even if communication between client and primary is disrupted in only one direction: if the client cannot contact the primary, it asks for a new one; if the primary is unable to reach the client, it breaks off communication, causing the client to request a new primary. The end result — a new client-primary attachment — is the same in either case.

3.3.4 Reaction to Partition

During a network partition, an initial `strict_read()` cannot be performed in a partition containing fewer than T-N+1 or a majority of secondaries, whichever is greater. T-N+1 secondaries are needed to locate the most recent version of the file, while a majority is needed to read and update the PCW and CT lists.

Similarly, when fewer than N secondaries are reachable, the `write()` operation will block if the client's cache fills due to the primary's inability to propagate updates to a sufficient number of secondaries.

3.3.5 Resolution of Conflicting Updates

In general, the most recent version of a file, as determined by timestamp, is the one that will be retained. Timestamps are assigned by clients, but since clients are not trusted, timestamps are validated by primaries during each pickup. If the client-assigned timestamp falls outside the range of plausibility, the primary assigns its own timestamp and informs the client of the correct time for future use. As a minimum, files stamped with a time in the *future* must have their timestamps corrected: any timestamp in the past is at least physically possible, but an update that exists before the alleged time of its birth is logically untenable.

Even when `strict_read()` is used exclusively, conflicts can still occur. For example, two clients may strictly read the same file, and then make conflicting updates into their caches. Ideally, if client A updates a file while client B is modifying its cached copy, client B should eventually learn that its updates were based on stale data. Client B's updates should not be written to secondaries without explicit authorization from B when such a conflict occurs. Conflict detection is particularly important after a network partition heals: we do not want updates to be overwritten with inconsistent versions that happen to have later timestamps.

We detect conflicts in our system through the use of timestamps augmented with a unique identifier (such as an IP address) that indicates which client last modified the replica in question.[4] After a read or a successful write (signified by

[4] Although *version vectors* [42] are a more powerful conflict detection mechanism than timestamps, the added power would not provide a significant advantage in our system. Version vectors are useful for detecting conflicts in an *optimistic* system such as Coda [51], which always returns the best value it can find. In contrast, our `strict_read()` operation will not return any value unless a certain level of consistency can be assured, thereby preventing many potential conflicts from occurring. Since the use of version vectors requires synchronously executing a protocol to

receipt of a purge notice), the timestamp/client pair associated with that version of the file is retained by the client. On future writes, if the service determines — while updates are being propagated to secondaries — that a newer version *written by a different client* has superseded the cached copy, a message concerning the conflict is sent to the client currently attempting to write. (The services escapes to a separate message-delivery mechanism in order to notify the client.) This is why a write quorum must consist of at least a majority of secondaries: conflicting updates to the same file must overlap at one or more secondaries.

There is no guarantee that the conflict message will arrive at the client, but this is not a problem. In the worst case, another message will be sent during the next round of pickup and propagation. It would be desirable to avoid the need for messages altogether, but since conflicting versions can be arbitrarily different from each other, there does not appear to be a general mechanism for resolving conflicts automatically (except, perhaps, in certain restricted cases such as logs and typed files). Kumar [27] has developed a scheme for use in the Coda file system [51] that can automatically resolve conflicting updates to *directories*, in many instances, but human intervention is still required in the general case.

Because of our asynchronous update propagation, a client must maintain *two* timestamp/client pairs for each file in its cache: a *base* timestamp, indicating the version upon which updates are based, and an *eventual* timestamp, which is the ID that will be assigned to the current version after propagation is complete. If a client is not a PCW for a given file, the base and eventual timestamps are identical. The eventual timestamp is given out to clients who read a file from a PCW's cache during an initial strict read: this is the version on which other clients' updates will be based, whether or not that version has been fully propagated.

The client ID is necessary to detect conflicts because no order is mandated for update propagation. Without knowing which client is responsible for an update, it would not be possible to tell the difference between out-of-order updates from a single client (which is fine), and out-of-order conflicting updates from multiple

exchange and update vectors, we ruled out their use.

clients (which must be flagged).

It is possible for an irresponsible client to *loosely* read a file, modify it, and write it back. However, one of the central themes of our work is that the cost of consistency should be borne by the users who demand it. If a client with write access to a file refuses to use `strict_read()` before modifying the file, that's the client's problem. A user who refuses to bear the cost of consistency is not guaranteed to receive it. Since write access is (presumably) granted only to clients who are willing to take responsibility for modifying a file properly, conflicts resulting from this problem should be very rare in practice.

This approach seems to conflict with the position adopted in Section 3.2.4: namely, that clients are not trusted and therefore must not be allowed to modify secondaries directly. If we do not trust clients to modify replicas properly, why trust them to use `strict_read()` in a sensible fashion? The answer is subtle, and rests on the difference between *data consistency* and *filesystem consistency*. In matters of data consistency, we provide the mechanism, but do not enforce it. Malicious or stupid clients with write access to a file can cause exactly the same problems in our system that they can cause in a centralized UNIX system: data may be intentionally or unintentionally overwritten.

It is crucial, however, that the file system itself have a consistent view of each file. It would be unacceptable, for example, for the system to return five different versions of a file on five consecutive `strict_read()` calls when no update activity is taking place. This could happen if a devious client wrote a different version to every secondary, all of which had the same timestamp. Therefore, in order to ensure filesystem consistency — a unique internal view of each file — replica management must be handled by trusted portions of the system.

It is worthwhile to note that the worst consistency problems an untrusted client can create are technically equivalent to a write following a loose read. In all cases, data of questionable origin and consistency is written into a file to which the client has write access. The service cannot guarantee the consistency of this data, but it can ensure that all clients who strictly read the file will receive the same data. Note that this is true even if a client deliberately provides a false timestamp: in the worst case, some other client may receive a spurious conflict message, but this is also possible during normal operations. We ensure that all conflicts are detected, but we do not guarantee that every potential conflict we detect will turn out to be a bona fide conflict.

3.3.6 Accommodation of Wandering Users

Finally, there is a case — unique to mobile operation — that we call the *wandering user*. Suppose a client strictly reads a file just before a network partition occurs. The client then makes several updates to the file, which are propagated to all reachable secondaries. Before the partition heals, the client wanders into a different partition and makes more updates to the same file, which are propagated to a different subset of secondaries.

This appears to be potentially chaotic, but actually works out quite well thanks to the timestamp mechanism. The client can make as many updates as desired during the partition because unavailable replicas do not cause write operations to block. When the partition heals, timestamps make it easy to decide which versions the secondaries should retain. If several wandering users make updates to the same file during a partition, the conflict detection scheme described above will sort out the divergent versions: the client ID attached to each timestamp makes it possible to impose a total ordering on the updates.

3.3.7 Semantics

Having reviewed the operation of our design during both normal and failure conditions, we now summarize the semantics of the three operations.

- The value returned by `loose_read()` is unpredictable, as is the fate of a write following a loose read.

- When a `write()` operation following a strict read returns, the client is assured that, if it remains reachable long enough, then its update will be stored at N secondaries, and — if it has not been superseded by a more recent conflicting update — will be installed as the latest value.

- In the presence of an arbitrary combination of writes by both loose readers and strict readers, the value returned by `strict_read()` is unpredictable.

- In the presence of only writes and strict reads, a strict read operation will return the value which is the latest among:

  1. The latest value written by a strict reader and present at any secondary server. The propagation of values to secondaries is subject to variation depending on the pickup schedules of primaries.
  2. The latest value in the cache of any reachable client which is a PCW.

  When there are no failures (all PCWs and at least a read quorum of secondaries are reachable), `strict_read()` returns the "latest" value; when there are failures, this fact can be detected and returned to the user.

3.4 Experimental Results

In this section, we describe the results of experiments designed to determine the constraints imposed by operating circumstances.

3.4.1 Experiments

Our motivation is to reduce or eliminate synchronous global communication from the typical case of common file service operations. Our idea of lazy propagation of updates and the corresponding "dual read call" interface leads to three hypotheses.

First, we hypothesize that few applications need what is provably the most recent version of a file. In a "personal workstation" setting, most updates come from a single source, and local cached copies at that source will therefore be up-to-date. The second hypothesis is that it is not hard for application programmers to know they need the latest value and hence must use the `strict_read()` call. A third hypothesis is that lazy update can take place quickly enough so that those applications that perform a loose read will typically receive the most recent value anyway.

We have performed experiments to test our hypotheses:

> Experiment 1: Measure use of existing file systems to determine the distribution of time intervals between write and subsequent read.

This information tells how fast update propagation should be. It also obliquely hints at the extent to which strict read is needed.

> Experiment 2: Determine what fraction of applications can be said to "obviously" require strict read.

Evaluating the `loose_read`/`strict_read` interface is an ease of use question, best answered by a prototype file system. In the absence of a full prototype, this experiment gives some information about how difficult to use the dual call interface might be.

> Experiment 3: Determine client update activity.

This information helps describe what size the cache should be and what pickup interval is appropriate.

3.4.2 Results

Our results were obtained by examining file traces gathered at Columbia. Traces were gathered on a Sun 3/80 running SunOS 4.0.3c. This version of SunOS offers a "C2 secure computing facility" that includes the ability to produce a system call audit trail. Using this feature, we gathered three large traces of a single user's file access activity. The traces report the file access activity of a volunteer user performing his normal work activity over a period of two weeks. The first trace contains 10,182 events captured over 33 hours. The second trace contains 12,472 events captured over 72 hours, while the numbers for the third trace are 25,440 and 86, respectively. During these hours activity varied widely and included compilations, document production, data analysis and display, large searches for certain files (i.e., UNIX "find" commands), USENET news reading, printing, and other operations.

Experiment 1: Write-Read Separation

When files are write-shared — that is, sequentially or simultaneously written by one client and read by others — it is important to know how closely an access follows an update. If updates to a shared file are widely spaced, a client is more likely to get the most recent version of a file even with a loose read. We analyzed our traces (which include the time at which a file was last modified) to determine the distribution of time between an update and the next open of the same file. Most files had long intervals between update and open, as indicated in the first three columns of Table 3.3.

It would be enlightening to know what fraction of writes and reads come from the same site. Unfortunately, we cannot use our data for such an experiment because it was garnered from the activity of a single user. Thanks to currency tokens, update propagation would in many cases not be required to catch write-read dependencies between applications at the same site. Therefore, the figures in the table overestimate the need for timely update propagation.

Experiment 2: Choosing Between Strict and Loose Read

The fourth column of Table 3.3 shows the percentage of those accesses made by applications that "obviously require" 1USR. This estimate was made by us based on our limited knowledge of a handful of applications appearing in the traces, and therefore constitutes an underestimate of the ease with which the need for `strict_read()` can be judged. Files we counted as obviously being used in a serial fashion included the intermediate files of a document processor and object files produced by the C compiler and passed onto the linker. While this eyeball technique is hardly scientific, it mimics the decision programmers would have to make.

The conclusion is that files with short update-open intervals were used by

Trace	Interval	Pct Accesses	Pct Known 1USR
1	0-3	1.4	100.0
	0-10	3.5	100.0
	0-60	6.8	98.5
	0-300	9.0	96.3
	0-600	11.4	94.3
	0-3600	17.4	87.6
	0-∞	100.0	38.1
2	0-3	2.0	97.6
	0-10	5.2	96.0
	0-60	9.7	95.7
	0-300	13.0	95.7
	0-600	16.5	95.6
	0-3600	23.9	95.1
	0-∞	100.0	47.1
3	0-3	2.0	98.2
	0-10	5.6	96.4
	0-60	10.8	90.1
	0-300	14.3	85.8
	0-600	17.9	82.6
	0-3600	26.0	77.2
	0-∞	100.0	39.5

Table 3.3: Update-Open Intervals

The interval is measured in seconds. The "Pct Accesses" column indicates what percentage of the write-read dependencies occurred during the given time interval. The numbers in the "Known 1USR" column indicate what percentage of dependencies during each time period could easily be seen to require strict_read().

programs that could easily know that serializability is required. In other words, files either had "safe" update-open intervals, or were being used by programs that understood the serial nature of what they were doing.

Experiment 3: Update Activity and File Lifetimes

Update Activity. Due to the limitations of our trace data, it was difficult to estimate the amount of update activity performed by a user. Ousterhout's 1985 study [39] found that each user read or wrote an average of 300 to 600 bytes of file data per second. This study was conducted on VAX/11-780s and is now outdated. Our own experiments, conducted on a Sun-3, suggest a higher figure — perhaps 1000 bytes per second. Both figures represent the total volume of reads and writes together, and therefore constitute a very conservative upper bound on the amount of update activity.

A rule of thumb is that reads outnumber writes by 2:1, so update activity can be estimated as perhaps 350 bytes/second. But even 1000 bytes/second of updates would be acceptable: this amounts to 60KB/minute, which suggests that even the small caches of portable workstations would be capable of storing several minutes of recent updates. Furthermore, a file may be updated several times between pickups, but only a single version need be maintained in the cache. This further reduces the actual amount of client cache space required to store updates between pickups.

File Lifetimes. Several studies have already been performed on file lifetimes [14, 39, 46, 49], and our own data confirms one of the earlier results: if a file is short-lived, it is usually very short-lived. The majority of files that were both created and destroyed during our traces have a lifetime of less than five minutes. Furthermore, the great majority of these files live for less than three seconds. Table 3.4 summarizes the results of our experiments in this area.

Taken together, the results of Experiments 2 and 3 suggest that:

1. The portion of the client's cache devoted to holding updates need not be much more than a few hundred kilobytes.

2. The conflicting goals of quick update propagation and overhead reduction due

Trace	Files	Pct 0-3	Pct 3-300	Pct >300
1	67	88.1	10.4	1.5
2	39	87.2	5.1	7.7
3	537	75.8	16.4	7.8

Table 3.4: File Lifetimes
This table lists the number of files that were both created and deleted within the trace. In all three traces, more than 75% of those files lived 3 seconds or less.

to caching can be well satisfied with a wide range of pickup intervals, starting at approximately 3 seconds.

3.5 Prototype Implementation

We have implemented a prototype version of our file system, building it on top of the Network File System (NFS) [24, 48] under Mach 2.5b [1], making use of the C Threads package [10]. This section describes the design of the prototype and the limitations of the current implementation, which implements most (but not all of) the full design. To simplify discussion, we will use the acronym "SBW" (Server-Based Writes) to refer to our system.

The design of the prototype may seem inelegant, and any evaluation from a performance standpoint would certainly yield results inferior to those we would expect from a full-fledged implementation. The prototype, however, is not intended to be an ideal implementation, but is instead a relatively quick and easy way to meet two goals:

1. Debug the design and major algorithms of our file service. As it turned out, we were pleasantly surprised by how few problems we encountered while programming; the current prototype was coded in less than five person-months.

2. Gain a tool to help judge the usability of the dual-read-call interface. (The current implementation is too primitive to provide any help in this area, but an implementation of the full design will provide us with the tool we desire.)

The relative ease with which we were able to put together the prototype, combined with the experimental data in Section 3.4, provide encouraging support for our ideas and design.

3.5.1 Overview

Several ideas shaped the design of the prototype:

1. Keep the number of kernel modifications to a minimum.

2. Use NFS to handle naming and data movement.

3. Use the `/tmp/` directory as the client's cache, with standard names distinguishing files.

Figure 3.5 shows the logical structure of the prototype. The flow of control runs as follows: trap critical file system calls within NFS, send the relevant data to a user-level daemon through an out-of-kernel socket, and wait for a new filename to be returned by the daemon. When this new filename is used instead of the one originally specified by the caller to NFS, the consistency controls provided by the SBW design are guaranteed to be enforced. In order to provide both SBW and non-SBW file access, SBW files must be referred to by a name beginning with `/sbw/`.

Standard `open()` calls are processed as follows: try `strict_read()`; if this fails, try `loose_read()`. If the client wishes to specify strict or loose read explicitly, this information is embedded in the filename: `/sbw/S/fname` for strict and `/sbw/L/fname` for loose.

Following the example in Figure 3.5: a user opens the file `/sbw/S/fname` for reading. At the start of the `open()` call, the relevant information is passed out of the kernel to the SBW daemon. The daemon, using its own data structures and

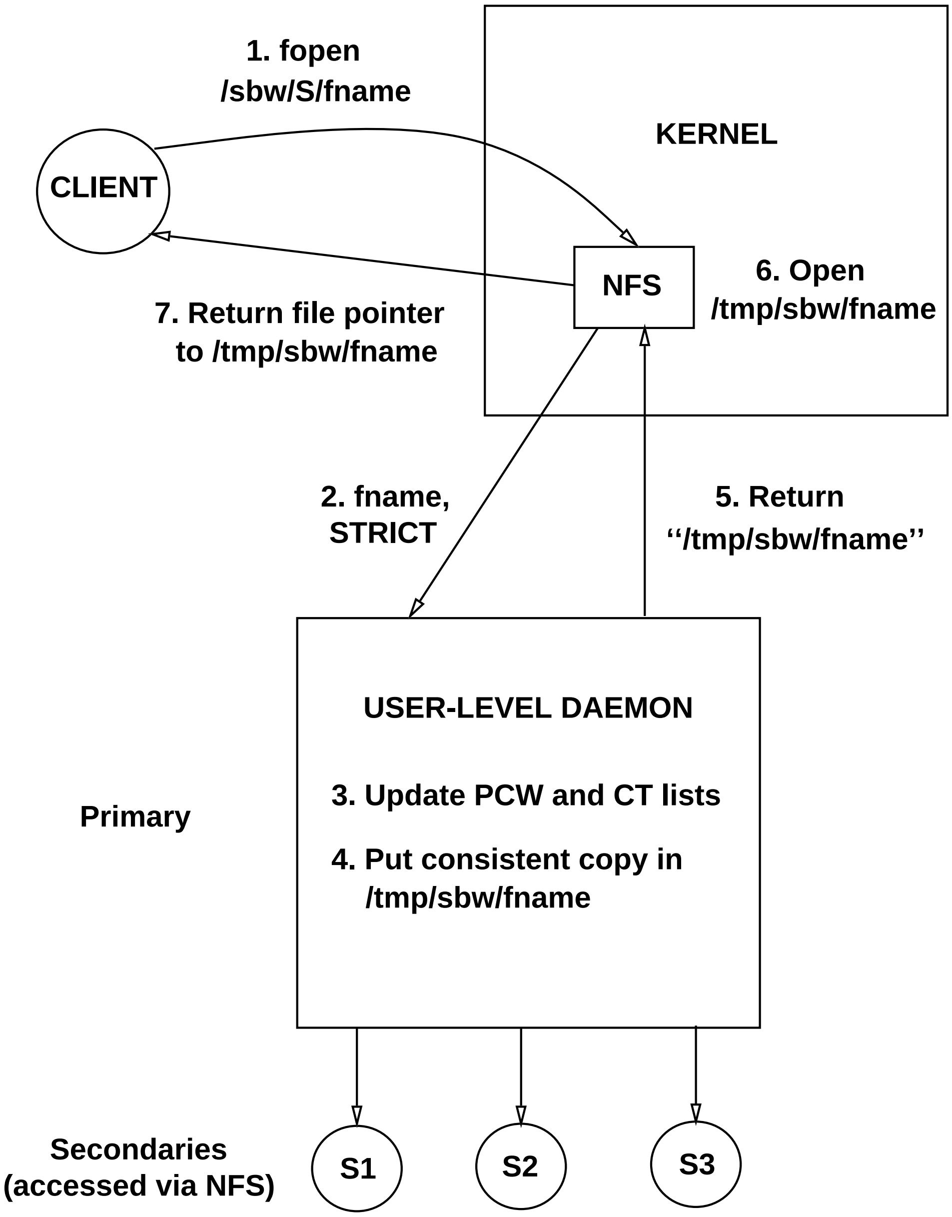

Figure 3.5: Logical Structure of Prototype

replication information, moves an appropriately consistent copy of `fname` into the `/tmp/sbw/` subdirectory, which acts as the client's cache. The daemon then returns the filename `/tmp/sbw/fname` to the kernel. The `open()` call continues on its way, using `/tmp/sbw/fname` instead of the name originally specified.

The prototype is predicated on the notion of using NFS to handle the low-level details of file accesses. This is especially important in the remote case, where we use standard NFS pathnames to access remote files comfortably. Reading or writing the `/tmp/sbw/` cache of a remote client is no harder: we simply mount `/tmp/` of remote clients locally.

As described so far, the daemon is equivalent to the primary server. However, centralizing power in a primary daemon that is closely tied to one particular client is both unrealistic and not fault-tolerant. We therefore introduce a process known as the *local file manager* (LFM), which runs at each client and each secondary. Rather than modifying a client's or secondary's files directly, the primary sends a Mach message to the appropriate LFM asking that the given operation be performed. In addition to decentralizing the primary's power — thus making replacement of the primary much simpler — this approach sets up clear boundaries for enforcement of authentication and security. Furthermore, LFMs provide the groundwork for a generic file manipulation interface similar to the OBJ interface in QuickSilver [56]. The primary server uses site-independent, generic commands when communicating with an LFM; the LFM handles all the site-specific details of file management at the given client or secondary.

An LFM runs on each client to handle file operations requested by its own primary or by other clients' primaries. In addition to the LFM, each client must have a small user-level daemon connected to its kernel via a socket. This daemon is responsible for reading client-initiated SBW operations out of the socket, and forwarding these operations (using Mach messages) to the primary for processing. The daemon is not directly responsible for any file handling; it is essentially a collection of function stubs. The actual primary may be located on any machine. The overall structure of the implementation is shown in Figure 3.6.

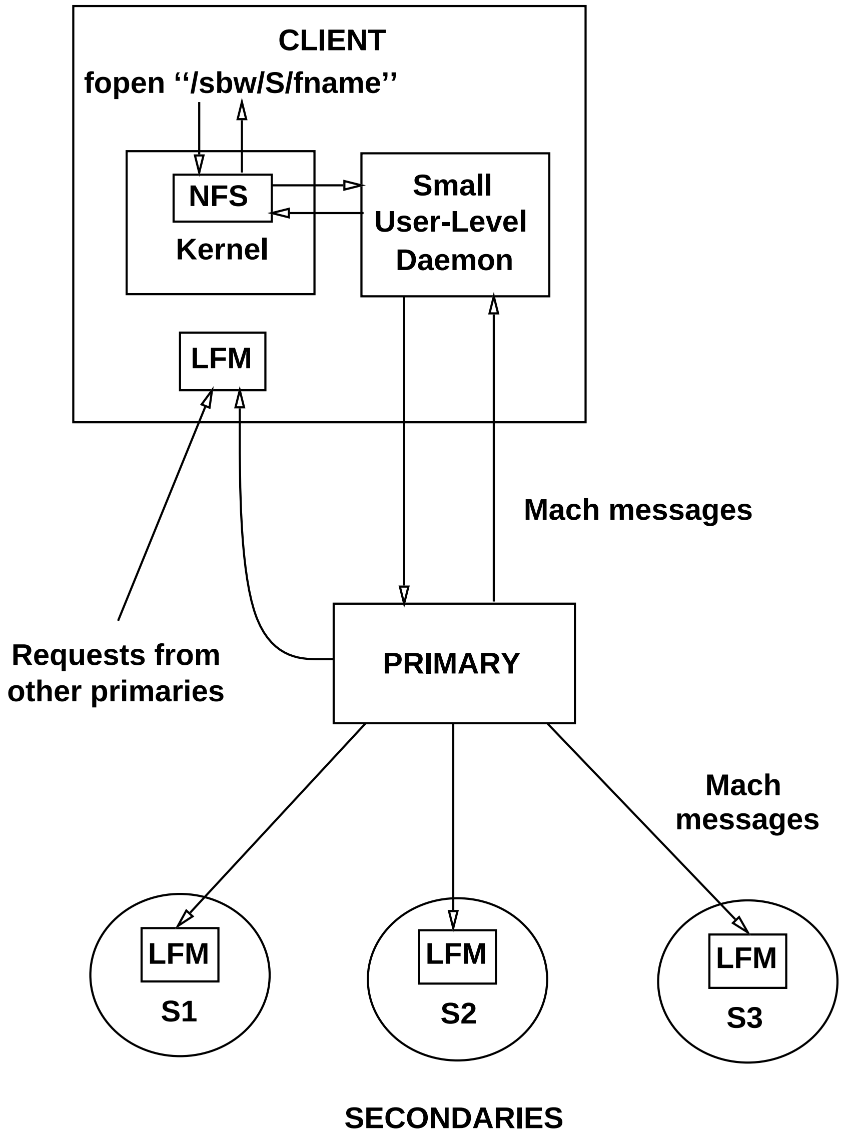

Figure 3.6: Implementation of Prototype

3.5.2 Source Code

The prototype consists of approximately 4300 lines of C. The code falls into three categories: modifications to the kernel (200 lines), kernel/daemon interface code (1200 lines), and daemon code (2900 lines). The current implementation centralizes all of the primary's power in a single user-level daemon for simplicity; local file managers are not yet implemented (but work is already underway).

Kernel Modifications

The SBW prototype requires very few modifications to existing kernel code.

- **vfs/vfs_syscalls.c:** Almost all of the changes are in this file, which handles system calls for operations on files other than `read`, `write`, and `ioctl` (whose arguments are file descriptors rather than filenames). For each system call, code has been inserted to call out to the daemon, and to use the new filename returned.

- **bsd/kern_exec.c:** This file must be modified to handle an `exec` or `execve` of an SBW file. This completes our series of "hooks" into the principal system calls with filename arguments.

Interface Code

At the heart of the interface code is a set of files that provides communication between the kernel and a user-level process via an out-of-kernel socket. **ksocko.c** and **ksocko.h** contain the kernel-side code; **socko.c** and **socko.h** are the files required on the user side. By providing this communication path, we have effectively dekernelized the major portion of the prototype.

The socket code supports buffered two-way communication. Sending messages out of the kernel is relatively simple: we spin off a C Thread on the user side to read data out of the socket as it arrives, and the user-level process reads from the buffer being filled by this thread. It is possible to declare a socket as unbuffered — perhaps for use when message processing is purely synchronous — but this can lead to unrecoverable errors if the user-level process fails to keep up with the messages that the kernel pushes into the socket. A buffered socket allows kernel data to be processed asynchronously, if desired; this feature will be critical in the file prefetching work described in Chapter 4.

In the SBW prototype, we need two-way communication between the kernel and the user-level daemon. In addition, we may have several operations simultaneously waiting for responses from the daemon, and this necessitates a general rendezvous scheme for message passing.

The rendezvous code is hidden within the `ksockWait()` function in ksocko.c. Messages are stamped with a unique ID, and `ksockWait()` sleeps until a response with the requested ID is received. In order to wake up sleeping processes, we must have a thread within the kernel that posts responses as they are received. This function is performed by `ksockReadDaemon()`, a function that never returns as long as the socket on which it was called remains open.

Since C Threads are not supported within the kernel, a thread must be spun off on the user side that is tied to the `ksockReadDaemon()` call. In the SBW prototype, this mechanism is provided in the new file **sbw.c** by a new system call. When the user-side daemon starts up, it immediately spins off a C Thread whose entire job is to make this new system call — a call that returns only when the prototype shuts down.

Supporting the message-passing thread is only a secondary function of **sbw.c**. The module's main tasks are to construct and transmit messages, and to parse responses that are received from the daemon. Also in **sbw.c** is another new system call to enable and disable the SBW filesystem. (There is a corresponding header file **sbw.h**.)

Daemon Code

The user-level daemon is responsible for implementing the consistency protocols specified by the SBW design. The daemon consists of several source files:[5]

[5]Harry Harjono coded the following source files: ops.c, findfile.c, multi_copy.c, fast_copy.c, and pickup.daemon.c.

- **disp.c:** The dispatcher. Reads requests out of the socket and spins off a C Thread to perform each function. The dispatcher is also responsible for reading in the file `MS.list` to set up the global list of secondaries. (`MS.list` is designed to simulate the information one would receive from the mapping server.)

- **ops.c:** Top-level code for SBW file operations. The dispatcher spins off threads that execute functions in this module.

- **quor.c:** Quorum-based operations, including all of the routines used to read, search, and modify PCW and CT lists. Includes a general mechanism for performing any operation on a quorum of secondaries.

- **time.c:** Timeout-based operations to avoid blocking during NFS calls.

- **sbw_user.h:** User-side data structures.

- **sbw.h:** Definitions used by kernel *and* user sides.

- **findfile.c:** Gathers a read quorum to find the most up-to-date replica of a file.

- **multi_copy.c:** Copies a file to multiple secondaries.

- **fast_copy.c:** The optimistically-named `fast_copy()` function replaces the shell command `cp`, which cannot be safely invoked from within a C Thread.

- **pickup.daemon.c:** This file is not linked into the daemon, though it is logically part of the primary server. The pickup daemon is a separate process that periodically picks up files from `/tmp/sbw/` and writes them to at least a majority of secondaries. The file `HOSTS.list` specifies which clients the daemon is servicing.

3.5.3 Restrictions

Although there are a number of features that have not yet been implemented, only one is a permanent restriction: the loose vs. strict decision must be made at the level of `open()`, not individual `read()` operations. This restriction is forced on us by the basic design: we provide an alternative filename at `open()` time, and from that point on, all NFS operations are performed on a vnode (the NFS analogue of an inode) associated with the substitute filename. As discussed in Section 3.2.2, however, it makes little difference in practice whether the loose/strict decision is tied to `read()` or `open()`, so we do not anticipate that this restriction will cause any problems.

3.6 Comparison with Related Work

For purposes of comparison, we have chosen three well-known and quite different working implementations: Coda [51], Deceit [53], and Echo [30]. Of the three, Echo enforces the strictest controls and provides the cleanest semantics (1SR). Its antithesis is Coda, a system that reads from and writes to whichever servers are available; version vectors are used to detect conflicts created by making multiple uncoordinated updates. (Coda is of particular interest because it was tailored to support mobile computing.) Deceit lies in between Echo and Coda, and is distinguished by the extent to which users can, on a per-file basis, vary the parameters controlling the consistency/availability tradeoff. In the following sections, we give a qualitative comparison of these systems and our own using two important criteria.

3.6.1 Performance

Coda is lazy to the extent that reads and writes are made into the client's cache, and users must wait only during open and close operations. An open request is processed at a "preferred server," but all reachable servers are contacted to ensure that the latest available copy is being returned. One of Coda's primary design goals was high scalability, and it succeeds. Involving the server only at open and close time is a great help. (There is some overhead at close time to update version vectors and check for divergences.)

Deceit allows unrestricted reads, but requires the client to obtain a write token before updating a file. This imposes additional overhead, but prevents divergent writes to a file. In addition, Deceit supports automatic file migration to a nearby server to improve performance. There is very little service disruption in Deceit, although a high "write safety level" requires the client to wait while multiple copies of a file are written. The default is to wait for just one copy to be written. Deceit's write tokens could cause scalability problems, but assuming a low degree of sharing — and a consequent lack of contention for tokens — this system should also scale well.

Echo takes the position that all writes should be visible immediately. This necessitates the use of both write and read tokens. Since sharing is infrequent, this is usually not a problem, but during periods of sharing, Echo's performance suffers to pay for its clean semantics: tokens must be shuttled back and forth between all clients who are accessing the file. Echo's clients must wait for writes to propagate — another cost of the nice semantics.

Our design can theoretically provide better write performance than any of the above systems, in the typical case. Ordinarily, a user's updates are made only to the local cache, and are picked up asynchronously by that user's primary server. In general, clients wait only during opens, since write and close operations are done locally and picked up later. In the worst case, a read operation will also require the client to wait, though typical strict reads and all loose reads are fast. Our scheme has the potential to scale very well. Because many files live for only a short time, they will never even be picked up by the primary server, and this will reduce network traffic.

3.6.2 Resiliency

With Coda, a client may both read and write so long as any replica is available. Coda takes the position that conflicts are sufficiently rare so that detection is preferable to prevention. This requires that a *reintegration* process of some complexity be run when a client reconnects after a period of disconnection. Server failures and recoveries are detected by periodically attempting to contact the servers storing replicas of a particular volume.

In Deceit, reads are always allowed. Writes may or may not be possible, depending on the setting of a parameter that allows generation of a new write token when the old one is unavailable. Recovery after a failure is moderately complicated, made more difficult by the fact that a server may crash while holding a write token.

In Echo, the goal of one-copy serializability prevents operation in a minority partition. Even a read operation requires that a majority of the replicas be up and in communication. Otherwise, a user might read stale data, which violates one-copy serializability. Recovery after a failure or partition is highly complex — much more so than in Coda or Deceit — and requires a period of operation dedicated exclusively to recovery.

Our design lies in between Coda and Deceit. Of course `loose_read()` is always possible when any replica is accessible, as in Coda. Writes are blocked only when the client's cache is full. The cache will fill up only if purge notices are not forthcoming. And purge notices will not be delivered if failures prevent an update from being sufficiently widely propagated. However, it is important to note that once service is re-established after a failure, the primary server can re-read any insufficiently propagated value from the client's cache. The client need not even be aware that any failure and recovery algorithms have been executed by the service.

The fault tolerance of `strict_read()` is dependent on usage patterns. In the worst case, the operation will block whenever a read quorum cannot be gathered. But by using CTs — which will usually be valid for long periods after an initial strict read — the typical case is very far from the worst case. Of all of these systems, Coda is the most tolerant of failures (though its recovery algorithms are non-trivial), and Echo is the least tolerant, with the additional difficulty of unusually complicated recovery procedures. Our design is at least as fault-tolerant as Deceit (usually more so), and has much the simplest recovery algorithms of any of the systems.

3.7 Conclusion

We have presented the details of a variable-consistency file service, emphasizing the dynamic relationships of clients, servers, and files. In addition, we have described and analyzed the recovery procedures for various types of failures, including the detection of conflicting updates. We have also presented empirical data to validate our ideas, compared our design to several existing systems, and described our prototype implementation.

Although mobile operation served as the initial motivation of our thinking, we think our proposal contains profound advantages for "regular" operation as well. The major novel features of our design are:

1. An extreme degree of client-service decoupling in the typical case. In all cases (except `write()` when the local cache is full), a client can simply give up on the current operation and begin again with a new primary server. This also reduces synchronous waits.

2. An interface in which the required consistency must be declared by the reader. (This is a consequence of the previous feature: the high degree of decoupling makes the "latest" copy time-consuming to locate; hence, we allow the client to decide if the expense is necessary.)

3. Currency tokens, which can drastically reduce the cost of strict reads after the overhead of the initial strict read.

The extra information provided to the service through the interface allows the implementation of both `write()` and `loose_read()` to be made completely lazy, and hence fast and scalable. Another major benefit is considerable simplification of the internal algorithms, especially those for recovery; very little state need be kept in non-volatile storage. These advantages, combined with the unusual degree of client-service decoupling, bode well for intermittently-connected mobile clients.

In essence, our design heaps the entire burden of implementing consistency onto the initial strict read. Thanks to the currency token — which persists until explicitly revoked, independent of cache occupancy — an initial strict read need be done very infrequently.

In our design a "primary server" is not a true replication site, but rather a bigger cache for reading and a coordinator for the asynchronous update propagation. By using the primary in this way, a client can be ignorant about the actual location of file replication sites. More important, though, is the elimination of blocking during typical write operations, courtesy of the beneficent primary.

We see two major drawbacks in this design: the need to program according to a new interface (although this can be mitigated by using one of the generic read calls suggested in Section 3.2.1), and the mediocre performance of the *initial* strict read. However, both our empirical data and our prototype implementation suggest that the benefits of this design substantially outweigh its liabilities.

4

Intelligent File Prefetching

4.1 Introduction

This chapter investigates how to automatically prefetch files from a file server to a client workstation. Another form of prefetching, prepaging, is an old idea. Prepaging has not had a major impact in computer architecture because of the tight time and complexity constraints on paging hardware and software. However, prefetching of files is a more promising endeavor for several reasons. First, since file accesses are less frequent than page accesses, the speed with which the decision to prefetch must be made is not so much of the essence. Further, the penalty for faulty prefetching is not so severe — wasting space in the client's file cache rather than in physical memory. Finally, the resource most needed to arrange intelligent file prefetching, namely client CPU cycles, is the resource most in excess in distributed systems now and in the likely future. In addition, we hypothesize that the work patterns of most individuals yield file access patterns that are quite pronounced and can be regarded as defining "working sets" of files used for particular applications. If the hypothesis is true, then it should be possible to capitalize on the promise of file prefetching.

We are particularly interested in the case where the link between client and server is low-bandwidth, unreliable, or both. Wireless mobile computing is an important subset of this case, in which typical bursty file access patterns often lead to poor performance due to insufficient bandwidth. Underpinning our thinking, however, is the notion that even a slow link may well provide adequate performance if we can find a way to spread out file accesses over time. By prefetching soon-to-be-needed files during slack periods when the link is not in use, we can create the illusion of a link with considerably higher bandwidth than is actually the case. As the ratio of processor speed to link speed increases — and given the current state of technology, such an increase will almost certainly occur — this argument grows in force. In the case of an unreliable connection, intelligent prefetching of files prior to disconnection can help clients get through the period in which they do not have access to a file server.

In Section 4.2 we describe an algorithm for automatically detecting an application's working set of files as it forms and then prefetching the working set later should the application be executed once again. Our method can in fact detect and exploit multiple distinct working sets generated by different executions of the same application. This capability is important when the same application is executed frequently but with different input; e.g., when a compiler is run on several different but similar modules during a system build. Successful file prefetching carries three major advantages, already hinted at above:

1. Applications run faster because they experience a higher cache hit rate.

2. There is less "burst" load placed on the network because prefetching is done in background rather than on demand.

3. Properly-loaded client caches can better survive network outages.

These advantages have extra impact if the client has a small cache and/or if the network is slow. Therefore, we expect our technique to be particularly useful for small portable workstations connected by a wireless network, which is typically very slow relative to modern wired networks.

There are two main costs of prefetching. The first is the cycles expended by the client in determining when and what to prefetch. Cycles are spent both on overhead in gathering the information necessary to make prefetch decisions, and on actually carrying out the prefetch. The second cost is the network and server bandwidth wasted when prefetch decisions inevitably prove less than perfect.

Section 4.3 reports results from simulations driven by real file access traces. In Section 4.4, we describe a working implementation and present performance data demonstrating its effectiveness. Thus, our technique is desirable (in small-cache environments), effective, and practical.

4.2 The Algorithm

In UNIX-style operating systems, every program gives rise to a tree of forked (child) processes, and all of these programs access (open or create) some files. Because the same files may be accessed or executed by many different programs, the graph that results will not necessarily be a pure tree: it may be riddled with cross-edges and back-edges. For ease of expression, however, we will refer to this tree-like structure as "a tree."

Our algorithm is based on a simple idea: if we save the distinct trees generated by each program, and if we compare trees being built by current activity with such saved trees, then we can detect which saved tree is being executed and prefetch the remainder of its files.

Many subtle problems creep into this seemingly simple algorithm. What happens if the saved tree is too large for the cache? What about the cross-edges and back-edges within "trees" that destroy the pure tree structure? We will describe how to address such practical details and argue that the resulting algorithm can be effectively used for file prefetching in a distributed file system.

4.2.1 Data Structure — The Working Forest

As a user does work, a graph called the *working forest* is built that reflects current file access patterns. Each file (program or data) is a node in the graph. Conceptually, there are two types of arcs:

1. If program A forks program B, we draw an arc from A to B.

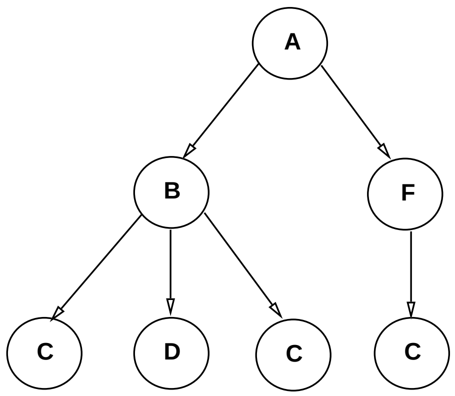

Figure 4.1: Sample Program Tree

2. If program A accesses or creates file B, we draw an arc from A to B.

Files A and B need not be distinct. In fact, we have observed many cases in which program A accesses itself as a data file. Although (1) and (2) represent distinct arc types, we have found that there is no need to distinguish between these types in the graph. We detect the two cases in different ways, but all edges are considered equivalent once they are in the graph.

In order to reflect the chronology of file accesses, arc order is preserved and we allow multiple arcs to exist from A to B. Consecutive accesses are ignored, but all non-consecutive accesses are preserved for use in a later phase of the algorithm. We do place an upper bound (currently 200) on the number of links emanating from a single node. Subsequent links from the same node are ignored. Since the working forest is constantly being pruned and rebuilt, 200 links are sufficient in virtually every case that we have seen. A form of mark-and-sweep garbage collection is periodically invoked to reclaim dead nodes in the working forest; the details of this process will be discussed in Section 4.2.7.

See Figure 4.1 for a diagram of the working forest after the following series of accesses:

1. Program A is executed

2. Program A forks program B
3. B accesses files C and D, in that order
4. B accesses C again
5. A forks F
6. F accesses C

Note that all of the references to file C actually draw arcs to the same node in the graph. Figure 4.1 shows the conceptual structure of the tree, not how it is actually represented as a graph in memory.

Because UNIX shells are invoked in such a wide variety of circumstances, they are treated differently from other programs. When a shell uses a file, we draw an arc from the closest non-shell ancestor of the given shell to the file now being accessed or executed. In essence, we draw arcs *through* shells, effectively cutting them out of the working forest. The justification for ignoring shells is that the shell, acting as a command interpreter, is simply an extension of the program that forked it.

Two more special cases need to be mentioned. Temporary files (whose names begin with `/tmp/` or `/usr/tmp/`) are never included in the working forest. By their nature, most of these files are not likely to occur in repeated patterns of computation — prefetching such files would likely be a waste. Further, devices ("files" whose names begin with `/dev/`) are not really conventional files at all, so they also have no place in the working forest.

4.2.2 Saving and Loading Trees

Whenever a program is executed, the working forest is checked for a tree (formed by a previous execution of the program) rooted at that program. If such a tree exists, it is copied from the working forest onto the top of a stack of trees saved for that program. The links emanating from the program in the working forest are then deleted. We allow multiple trees to be saved for each program, but in order to reduce storage requirements and to ensure that trees are not saved indefinitely, we place an upper bound (currently five; an arbitrary figure) on the number of saved trees. If a sixth tree is pushed onto the stack, the bottom tree on the stack is discarded.

The result of this process is that, when the N+1st execution of a program begins, the tree formed by the Nth execution is unlinked from the working forest and moved to that program's stack. The moment the N+1st execution begins is when the working set of the Nth execution is defined. Notice that while the moment of definition may seem "late," in fact it is the first time at which the working set information could be exploited — and the earliest time at which we can be sure the Nth execution is complete.

As the N+1st execution begins to form another tree within the working forest, its tree is compared to all trees saved on the stack. If it seems that the tree in the working forest is likely to match one of the saved trees, then prefetching of the saved tree is initiated.

In order to choose which tree, if any, to prefetch as a program executes, the algorithm follows a simple guideline: wait until the observed file usage pattern matches only one of the saved trees. If several trees match, wait for more file accesses. If no trees match, do not prefetch for this execution. In order for the forming tree in the working forest to match a saved tree, its initial file references must match exactly: all of the files that the program has directly touched so far must also have been touched by the root of the saved tree.

What happens if the guess is wrong? We may prefetch a saved tree, only to discover that it was a poor match for the files that were actually accessed next. So when the program executes the next time, we must compare the tree that was actually built with the tree that we prefetched. If the match is "close enough," we discard the saved tree and replace it with the tree we just built; we assume that newer is better, if the match was otherwise reasonable. But if the constructed tree is nothing like the tree we prefetched, we must save both of them on the stack of trees for that program.

The definition of "close enough" is a heuristic that is effective for the data we have seen: if at least 40% of one tree's files are found in the other tree (in any order), we deem the match to be successful. This allows for imperfect matches, which is what we want — rarely will two trees touch exactly the same files.

Figure 4.2 contains an example from one of our traces. There are two trees rooted by "cc," the UNIX C compiler. The first tree is responsible for creating object files; the second links object files into executables. They both start the same way: `cc` executes, and then reads a file containing language information. Our algorithm waits until the set of files that the current `cc` execution touches is found in exactly one of the saved trees. If `cc` should access some previously unknown file before a tree is chosen, we give up on prefetching for `cc` on this execution — apparently there is no matching tree.

Note that we do not consider access order important here. If, for some reason, `cc` had touched `ccom` before doing anything else, tree (a) would have been selected on the spot.

It is important to note that this entire process is repeated for each program within the tree that we have loaded. The idea is that each program prefetches its own tree, and the tree thus selected may or may not be a subtree of the larger tree we prefetched earlier. This effectively allows minor prefetch corrections to be made at every level in a tree, which reduces the cost of a bad guess at a high level. It also means that there is no concept of "the tree currently being prefetched": the prefetch set is constantly being modified as each constituent program is executed. This fact substantially eases implementation.

Our algorithm requires us to save multiple trees for each program. It would be desirable to compress this saved data down to a single tree, if possible; Driscoll et al. [12] have already proposed a method for storing multiple versions of data structures efficiently. Unfortunately, the very nature of our file trees makes them poor candidates for such compression. Because a file can appear at any point in any tree, we would need to store a large amount of contextual information at each node: "If the path we took to arrive at this node is such-and-such, then the possible children are X, Y, and Z; else, if the path is so-and-so then ...." It seems unlikely that we would achieve any appreciable savings in space by using such a cumbersome

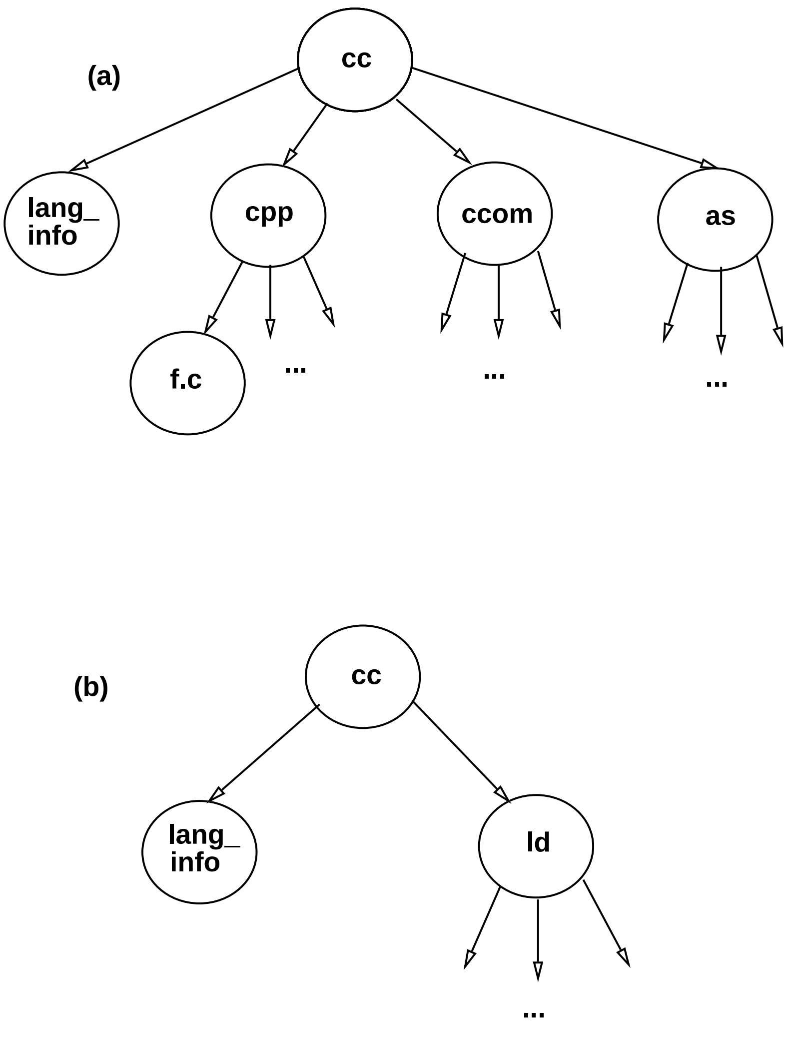

Figure 4.2: Two Different 'cc' Trees

method; time and complexity constraints also argue against this approach.

4.2.3 Tree Splitting

If the prefetched tree is too big to fit into the cache, we resort to a tree-splitting scheme. First, we step through the tree in preorder — the order in which files are likely to be used — until we have found a subtree that will fit comfortably into the cache.

Now for the hard part. Stepping through the tree again, we take all of the nodes and subtrees that were severed from the original tree, and point them directly to a single new root, thus creating a single subtree to be saved. (This is actually a form of path compression.) When a file is accessed that has one of these pointers to a saved subtree root, we load the subtree, splitting it again if necessary. Note that this does not take precedence over the tree-selection scheme described earlier; we load saved subtrees only when there is nothing else we can do.

Figure 4.3 shows how the tree in Figure 4.1 is broken up when using a cache that can hold three files. Note that D and C are pointed directly to the new root due to path compression. The lower half of Figure 4.3 shows what occurs when the resulting tree is split yet again.

4.2.4 Common Prefix Trees

Oftentimes the first few file accesses of all saved trees are the same. In this case, the algorithm will typically miss on every file access in this "common prefix." We have added an enhancement to the algorithm that avoids most of these unnecessary misses.

Assuming that there is no unique saved tree to load, and that there is no split subtree from a previous load available, then we immediately compute and prefetch a *common prefix tree* for the program. The prefix tree is a 2-level tree whose root is the program itself; the children are the root's file accesses that are common to all the saved trees. We require strict matching in this case: if the root of our prefix

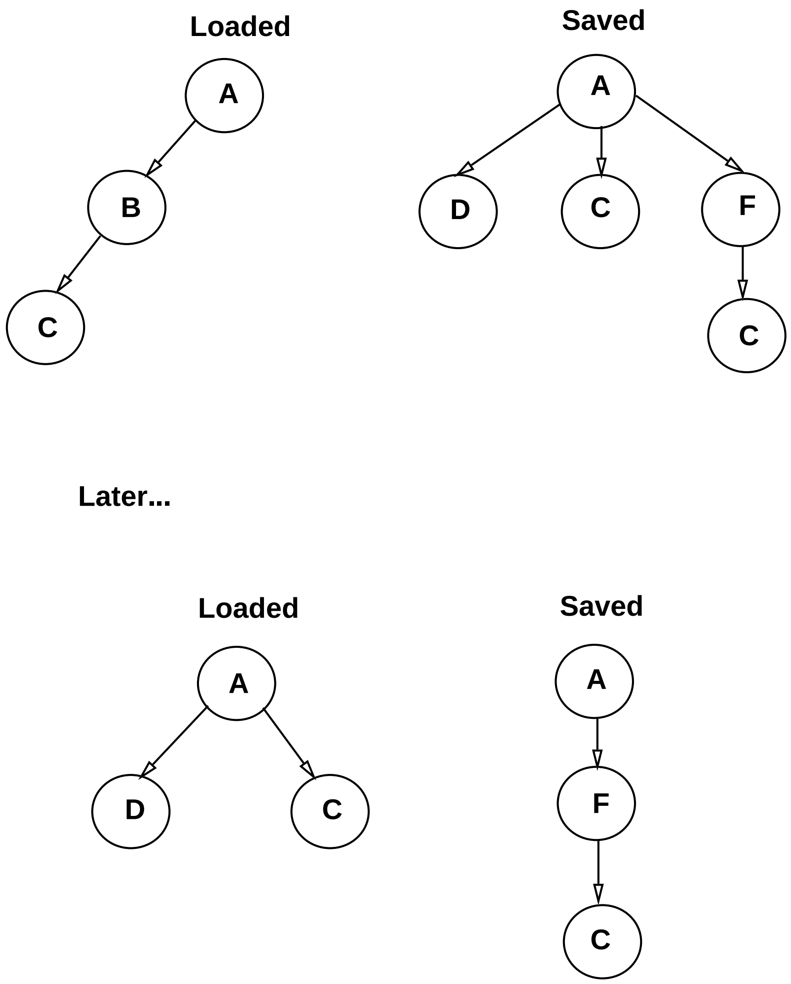

Figure 4.3: Tree Splitting

tree has N children, this means that the first N children of the root in every saved tree are identical.

Once the prefix tree is computed, it is loaded like any other tree, and then destroyed. To avoid the problem of tree splitting, our construction ensures that prefix trees are never too large for the cache: extra files in the common prefix are ignored.

4.2.5 Cycle Trees

A case requiring special attention is that of long-running programs that establish repeated file access patterns within a single execution. The basic algorithm cannot handle this case at all, since the tree-matching code is triggered only when a program begins an execution.

To get around this problem, we use *cycle trees*. Such a tree consists of a cyclic pattern of file accesses that has been detected within a running program. Our algorithm checks for cycles whenever it cannot load a tree of any other type. It compares the two most recent access patterns that begin with the file just touched, and builds a 2-level tree that includes the longest common prefix of the two patterns.

Cycle trees are very much like common prefix trees. They include only the immediate children of the program in question; they contain exactly those file accesses common to previous executions or access patterns; and by construction, they are never too large for the cache. They are loaded like any other tree, and destroyed immediately. Figure 4.4 shows how a cycle tree is constructed from a program's pattern of file accesses.

4.2.6 Prefetch Confidence

One final point about loading trees of any sort: in order to avoid destroying the entire cache due to a bad prefetching guess, we never fill more than 80% of the cache when loading a tree. The figure of 80% for "prefetch confidence" is arbitrary. Varying it has little effect on the hit rate unless the number is made small.

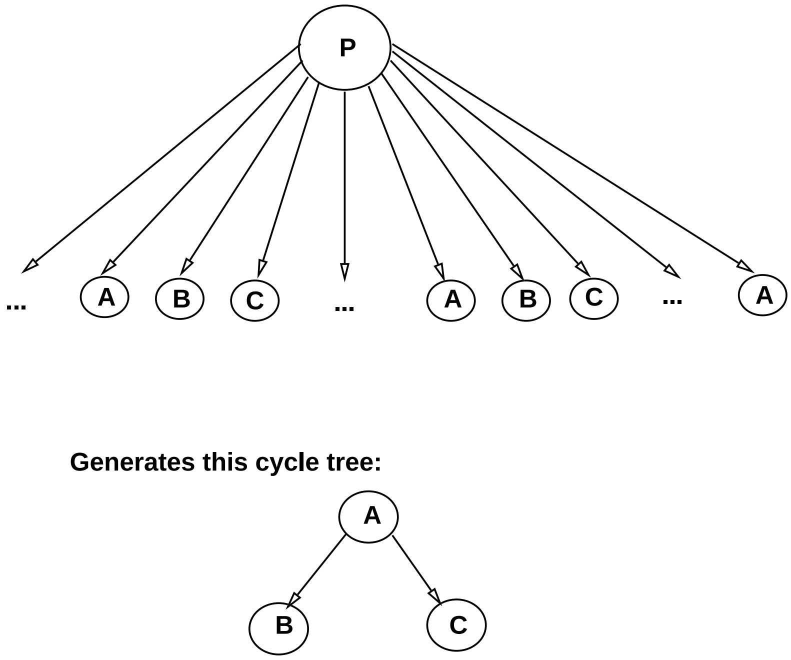

Figure 4.4: Cycle Tree Construction

4.2.7 Garbage Collection

In order to keep the number of nodes and saved trees from growing indefinitely, it is necessary to perform garbage collection at periodic intervals. Because an arbitrarily long time can elapse between executions of a program, however, no saved tree can ever be classified as "garbage" with absolute certainty. Instead, we adopt a straightforward heuristic: if a file has not been touched (opened or executed) since the last garbage collection, we simply *assume* that the file's associated trees and working forest nodes can be considered garbage. In the worst case, we will have to rebuild some trees if the program is executed again. Ordinarily, however, we will reduce the amount of storage required by the prefetching algorithm by retaining only those nodes and trees that are in active use.

In principle, the garbage collection algorithm is the traditional "mark and sweep" method. In the marking phase, nodes in the working forest that have been accessed since the previous garbage collection are marked as active along with their associated saved trees. The sweep phase examines all nodes and trees, reclaiming the storage of those that have not been marked active.

There are several unusual aspects of the algorithm, however. First, because our "trees" can actually contain back-edges and cross-edges, we must avoid getting stuck in cycles during the marking phase. Second, in order to gain any benefit from cycle trees, long-running programs must not be garbage collected. Whenever a cycle tree is loaded, its parent program's node in the working forest is explicitly touched to prevent this problem.

Third, due to the subtleties of tree splitting, particular care must be taken when marking a saved subtree. It is possible for a particular file to appear in *both* portions of a split tree, and if the split point occurs on one of these duplicated files, we have a unique situation in which the same file is represented by two distinct nodes in the saved subtree. This leads to disastrous results if the garbage collector blithely marks nodes based purely on filenames rather than node addresses: a portion of the saved subtree is not marked, and is therefore unintentionally garbage collected.

Finally, marking nodes is actually an iterative process akin to running a document through a text processor repeatedly until all cross-references are resolved. When a program is marked, all of the files in its saved trees must also be marked. This, in turn, requires that the saved trees of the newly-marked files be recursively subjected to the same process. The marking phase is not over until the set of marked nodes stabilizes, at which point all chains of saved trees have been completely searched.

4.3 Simulation Results

Our initial results were obtained by trace-driven simulation.

4.3.1 Trace Data

We used the same file traces as those previously described in the trace-driven experiments of Section 3.4. The monitored operations were those for process creation (`fork`, `execv/execve`) and file access and creation (`open`, `creat`, `mkdir`). These are the operations needed to build the working forest.

4.3.2 Simulation Methodology

The task of the algorithm simulator is to step through the trace files, managing its cache in accordance with the graphical machinations prescribed by our algorithm. Since our trace data provides no information concerning file sizes, we simply define cache size by number of files: for example, 20 files, not 200 kilobytes.

To make the simulation more realistic, we use a delay factor: a file is not assumed to be in the cache until one second after background prefetching is initiated. If the user accesses a file that is being prefetched before the delay period has elapsed, the access is treated as a cache miss. Assuming a one-second delay may seem excessive for small files routed over a fast local network, but the timestamps in our trace data are rounded to the nearest second, preventing us from using a smaller time interval.

As a basis for comparison, the simulator manages a separate cache for each of three algorithms other than the tree-based scheme. The first competing method is simple LRU replacement; the other two are more worthy opponents: *simple* prefetching schemes. If a trivial algorithm works well, why bother with the relative complexity of the tree method? The two simplistic prefetching techniques we evaluate are variants of the same idea:

- **Stupid pairs:** When file F is accessed, prefetch the file that was accessed immediately after F the last time F was opened.

  Sample series of accesses: F, G (remember F-G), F (prefetch G, remember G-F), H (remember F-H), F (prefetch H, remember H-F), H (cache hit).

- **Smart pairs:** Keep track of all files that are accessed immediately after F, and when F is accessed the next time, choose one according to the frequency distribution.

  Sample series of accesses (only F's pairs are shown): F, G (F-G1), F (prefetch G), H (F-[G1, H1]), F (prefetch G or H with 1:1 weighting), H (F-[G1, H2]), F (prefetch G or H with 1:2 weighting).

In our file traces, five particular files involved in the dynamic linking process are opened by every program that executes. These files are accessed so frequently that any reasonable method would certainly keep them locked in the cache. Since we want to see how our method compares with LRU and simplistic prefetching in the cases where the algorithms might reasonably be expected to differ, we filtered these heavily-used files out of our trace data — i.e., *before* the simulation. In order to present fair comparisons between our algorithm and the other methods despite this adjustment, we report *miss ratio* rather than hit ratio. The reason is that of the three significant numbers summarizing a simulation run (number of hits, number of misses, number of total accesses), only the number of misses is unaffected by removing certain accesses that are assumed to always hit in the cache. So the ratio of miss ratios of any two algorithms remains the same despite the adjusted data (because the ratio of miss ratios is simply the ratio of misses).

The current and parent directory files ('.' and '..') are also accessed very frequently. Although these names do not represent fixed files, it is a simple matter to cache the current and parent directories, and so these file references are filtered out as well. We are thus left with the interesting cases that provide the most reliable measure of our algorithm's effectiveness vis-a-vis the other methods under consideration.

In the simulation, file names are stored in graph nodes. Names are handled in a straightforward way: a file name is stored exactly as it appears in the trace data; there is no expansion of '.' or '..' prefixes. In addition, no attempt is made to recognize when multiple names are being used for a single file — each name becomes a separate node in the graph. Although this simplistic approach suffices for the simulation, the working implementation requires a more complex method of name resolution, and this will be described in Section 4.4.

4.3.3 Results

We expected that the increased intelligence of our method would be more effective versus the other methods in smaller caches. After all, in the extreme case of an infinitely large cache, it doesn't matter what "replacement" policy one uses, since no files are ever replaced. The "Optimal" column in Table 4.1 shows the miss ratios that one would obtain with an infinite cache. In practice, no algorithm can approach the optimal miss ratio because of the many repeated large `find` operations in our traces — and these operations make large portions of the traces antagonistic to tree-based prefetching.

Bearing all of that in mind, we ran our simulator using cache sizes of 20, 40, 100, and 1000 files. (Except for the 1000-file case, these sizes are quite small: if our algorithm failed to perform well using these caches, it would be unlikely to be effective in general.) We kept track of the cache misses for our tree-based algorithm and for its competitors. A summary of the results for the three traces appears in Table 4.1. The tree-based algorithm has a substantially better miss ratio than the other methods with a cache size of 20 files. Even with larger caches, however, the

Trace	Size	LRU	Smart	Stupid	Tree	Optimal	LRU/Tree	Stupid/Tree
1	20	58.9	55.2	53.3	44.9	18.0	1.31	1.19
	40	51.3	48.2	46.5	39.2		1.31	1.19
	100	34.2	32.3	31.2	28.9		1.18	1.08
	1000	20.7	20.1	20.1	19.5		1.07	1.03
2	20	67.2	60.1	57.6	48.2	14.9	1.40	1.20
	40	51.2	50.3	46.1	39.9		1.28	1.16
	100	37.8	33.5	33.7	33.7		1.12	1.00
	1000	21.7	20.0	20.1	20.0		1.09	1.01
3	20	55.5	50.8	48.8	43.1	13.3	1.29	1.13
	40	43.9	40.9	39.5	36.7		1.20	1.08
	100	33.4	30.4	30.1	29.5		1.13	1.02
	1000	20.8	19.1	19.3	18.9		1.10	1.02

Table 4.1: File Cache Miss Rate (percent)

tree method out-performs both LRU and the simplistic prefetching algorithms.

Surprisingly, the stupid pair scheme worked better than the smart pair approach when applied to our traces. This unexpected result can be explained by considering the locality of many accesses: often, a user works on one file or one group of files for some time (editing or compiling), then moves on to similar operations with different files. Stupid pairs are well-equipped to handle this usage pattern, since they invariably prefetch the most recently used successor file. The seemingly superior intelligence of smart pairs actually becomes a liability when locality is strong; files no longer in active use may be given undue weight if they were heavily accessed in the past. Indeed, the stupid pair approach worked well enough for us to add it to the tree algorithm: we prefetch the file prescribed by the stupid pair scheme in *addition* to any files specified by the tree method.

A logical extension of pair-based prefetching is a *trigram* scheme: use the previous *two* file accesses to predict the next one, or alternatively, prefetch two files instead of one based on the current file access. The performance of these approaches is shown in Table 4.2, along with the smart/stupid pair data for comparison. Using

Trace	Size	SMART Use 1 PF 1	SMART Use 2 PF 1	SMART Use 1 PF 2	STUPID Use 1 PF 1	STUPID Use 2 PF 1	STUPID Use 1 PF 2
1	20	55.2	55.9	49.0	53.3	54.4	47.8
	40	48.2	48.6	43.0	46.5	47.6	42.2
	100	32.3	32.3	30.4	31.2	32.0	28.9
	1000	20.1	20.2	19.4	20.1	20.3	19.6
2	20	60.1	59.2	50.7	57.6	57.7	49.5
	40	50.3	48.2	42.9	46.1	47.5	39.6
	100	33.5	33.7	29.7	33.7	33.7	30.1
	1000	20.0	20.3	21.2	20.1	20.3	19.0
3	20	50.8	49.3	44.1	48.8	48.7	42.4
	40	40.9	39.7	35.9	39.5	39.2	34.6
	100	30.4	30.2	27.5	30.1	29.9	27.1
	1000	19.1	19.4	17.6	19.3	19.4	18.1

Table 4.2: Pair and Trigram Miss Rates (percent)

more data to predict the next access usually does not help (and may actually hurt) in such a simplistic scheme: because matching must be exact, fewer prefetching requests are generated. However, prefetching two files instead of one is surprisingly successful, and works particularly well when the cache is large: although trigrams prefetch many files that are not used, bad prefetching guesses do not usually bump valuable files out of the cache in this case. The simulation, however, does not take the costs of increased network and server load into account. This can be deceptive in a scheme that prefetches a large number of files if the cache is large enough to bear the cost of incorrect prefetching decisions.

We ran two additional experiments comparing the tree method to LRU — no prefetching at all — and removed the one-second prefetching delay, since this figure is decidedly excessive in most cases. First, we monitored the burst hit ratios: how well each method did over each burst of 512 accesses. At the end of each run, we checked to see which method had won the most bursts. As shown in Table 4.3, the tree method won this comparison easily and consistently. This shows that the

Trace	Size	Wins	Losses	Ties
1	20	18	0	1
	40	18	0	1
	100	13	5	1
	1000	4	3	12
2	20	20	1	3
	40	18	2	4
	100	14	5	5
	1000	3	8	13
3	20	44	1	4
	40	37	6	6
	100	31	10	8
	1000	13	13	23

Table 4.3: Tree vs. LRU over Bursts of 512 Accesses

tree algorithm's superiority is steady and stable, and not simply the result of a few exceptionally fruitful prefetch sequences.

Ties at the burst level are attributable to a large number of new files suddenly being brought into the cache in both methods — as will happen with a large `find`, for example.

Second, we measured the miss rate during those periods immediately following a tree prefetch. After loading a tree of size N, we monitored the miss ratios of both methods for the next N-1 accesses (the root was not counted since it initiated the prefetch). Table 4.4 shows the results of these measurements. As desired, the miss rate for the tree method is quite low in the periods following prefetches, indicating that our algorithm is effective in choosing which files to prefetch.

The entire simulator consists of approximately 2200 lines of C. Table 4.5 gives the purpose, code size, and approximate running time of each of the major pieces of the simulator. Several of these routines are called quite infrequently, so two timing figures are given: milliseconds per call, and average milliseconds per file access when calling frequency is taken into account.

Determining if a prefetch succeeded occurs about 5 times in every 100 ac-

Trace	Size	LRU	Tree	LRU/Tree
1	20	56.9	15.5	3.66
	40	44.8	13.6	3.31
	100	18.1	8.8	2.06
	1000	8.5	6.7	1.27
2	20	62.5	12.8	4.87
	40	34.0	9.1	3.74
	100	10.4	8.1	1.28
	1000	8.7	6.6	1.32
3	20	45.9	17.7	2.60
	40	26.2	13.7	1.91
	100	12.9	9.5	1.36
	1000	6.5	5.7	1.16

Table 4.4: Miss Rates following Tree Prefetches

cesses. Tree splitting is rarer, and is only required about 75 times in 10,000 accesses. Garbage collection is performed once every 4096 accesses. The simulation was timed on a Sun 4/490, a 22 MIPS machine.

4.3.4 Limitations

Simulation results must always be interpreted carefully, and we have identified several ways in which our simulation does not accurately mirror a plausible real-life implementation:

1. References to a few commonly-used files are filtered from trace data.

2. Cache size is measured in files, not bytes.

3. The cache sizes are perhaps unrealistically small.

4. Added network and server load is not reflected in the simulation: prefetching latency is fixed at one second per file.

CODE SECTION	Lines	Call	Avg.
Add links to working forest	200	0.09	0.09
Tree selection for prefetch	470	1.45	1.45
Did prefetch succeed?	90	1.21	0.06
Tree splitting	190	2.56	0.02
Garbage collection	190	144.09	0.04
Utility routines	540	—	—
Simulator skeleton	380	—	—
Declarations and constants	110	—	—
TOTAL	2170	—	—

Table 4.5: Size and Running Time of Code Sections

5. Conversely, a one-second delay for background prefetching is probably a high estimate in the typical case; it is certainly high in a LAN environment.

6. References to certain other commonly-used files are ignored by our algorithm. Thus files in `/tmp` hit sometimes when the LRU method is used, but rarely when our method is used.

7. Our tree-based algorithm suffers from a learning curve. Running the algorithm twice in a row shows that, for our traces, the learning-curve penalty raises the miss rate by up to 2%.

The first four simulation inaccuracies tend to overvalue our algorithm, while the last three tend to undervalue it.

Our algorithm depends on a UNIX-style model of process forks in order to build trees. Fork information is vital to our method, since it is used to define a hierarchical chronology of file accesses, leading to the use of tree-based working sets of files. Our dependence on fork information mandates that the algorithm be run on the client side. This is not really a problem: in the interests of scalability, it is certainly desirable to minimize the use of server resources.

Certain file access patterns limit the effectiveness of our algorithm. In Trace 1, more than half of all misses were caused by a data analysis program that, across repeated executions, accessed different subsets of files within the same directory or else accessed the same files but in a different order. Our algorithm deals poorly with this case, though it would be possible to reduce or eliminate this problem by prefetching entire directories. However, directory usage measurements [14] do not make it clear that, in general, such a step would help more than hurt.

Our algorithm is implicitly based on whole-file caching. This is perhaps acceptable, since studies indicate that the great majority of files are read in their entirety [14, 39]. Furthermore, there is a minor trend toward file systems that use whole-file caching [21, 54]. Very large files and shared files should remain at the server, however, and we have not yet addressed the problem of how to deal with these files.

Finally, we would like to gather more trace data from different environments. The traces that we are currently using, however, provide encouraging evidence for the effectiveness of our algorithm.

4.4 Implementation

We have implemented a fully functional version of our algorithm. Like the file system prototype, the prefetcher is built on top of the Network File System (NFS) under Mach 2.5b, and makes use of the C Threads package. In this section, we will describe the design of the implementation, and evaluate its performance.

Our primary motivation for implementing the algorithm was to eliminate most of the simulation limitations discussed in Section 4.3.4. The learning curve is inherent in the algorithm, but the other limitations need not be present in an implementation. It is especially desirable to move beyond the simulation's file-based cache management and artificial handling of prefetching delays.

The implementation went through two major versions (the earlier version was completely within the kernel), and required a total of approximately ten person-months to complete — a non-trivial portion of which was spent unraveling the intricacies of NFS. The final implementation consists of 5050 lines of C: 150 lines of kernel modifications, 1200 lines of new kernel and interface code, and 3700 lines of user-level code. (800 lines of this total — the out-of-kernel socket code — was used in the file system prototype as well.)

4.4.1 Design Overview

Figure 4.5 shows the basic structure of the prefetching implementation. The design is similar to the file system prototype in that most of the interesting work is done by a user-level daemon, with critical information passed out of the kernel via a socket. Unlike the prototype, however, many changes had to be made to NFS in order to support prefetching.

To understand how the prefetching code works, it will be useful to follow the flow of data during a typical file operation. Suppose a user-level program opens the file “myfile”. The open request goes through NFS, which sends a message of the form “FILEREF: pid x, file ‘myfile’ ” to the prefetching daemon via an out-of kernel socket. This is an asynchronous operation; NFS does not wait for any acknowledgement of its message — in fact, the communication through the special socket is completely one-way in this code.

The daemon builds the working forest using the information it gets from NFS, following the algorithms described in the previous sections. When the daemon decides that a tree of files should be prefetched, it forks a lightweight thread for each file to be prefetched.

However, the prefetching threads cannot simply use `fopen("r")` to read files into the NFS cache, for two reasons:

1. Such reads would be indistinguishable from bona fide read requests generated by regular programs. Hence, we would go into infinite recursion as NFS continued to send us spurious file reference messages about files that were simply being prefetched.

2. We only want to read each prefetched file into the NFS cache, not into user space or any intermediate buffers. The default sequence of buffer copies when a remote file is opened with NFS is: remote file, NFS buffer, per-file buffer,

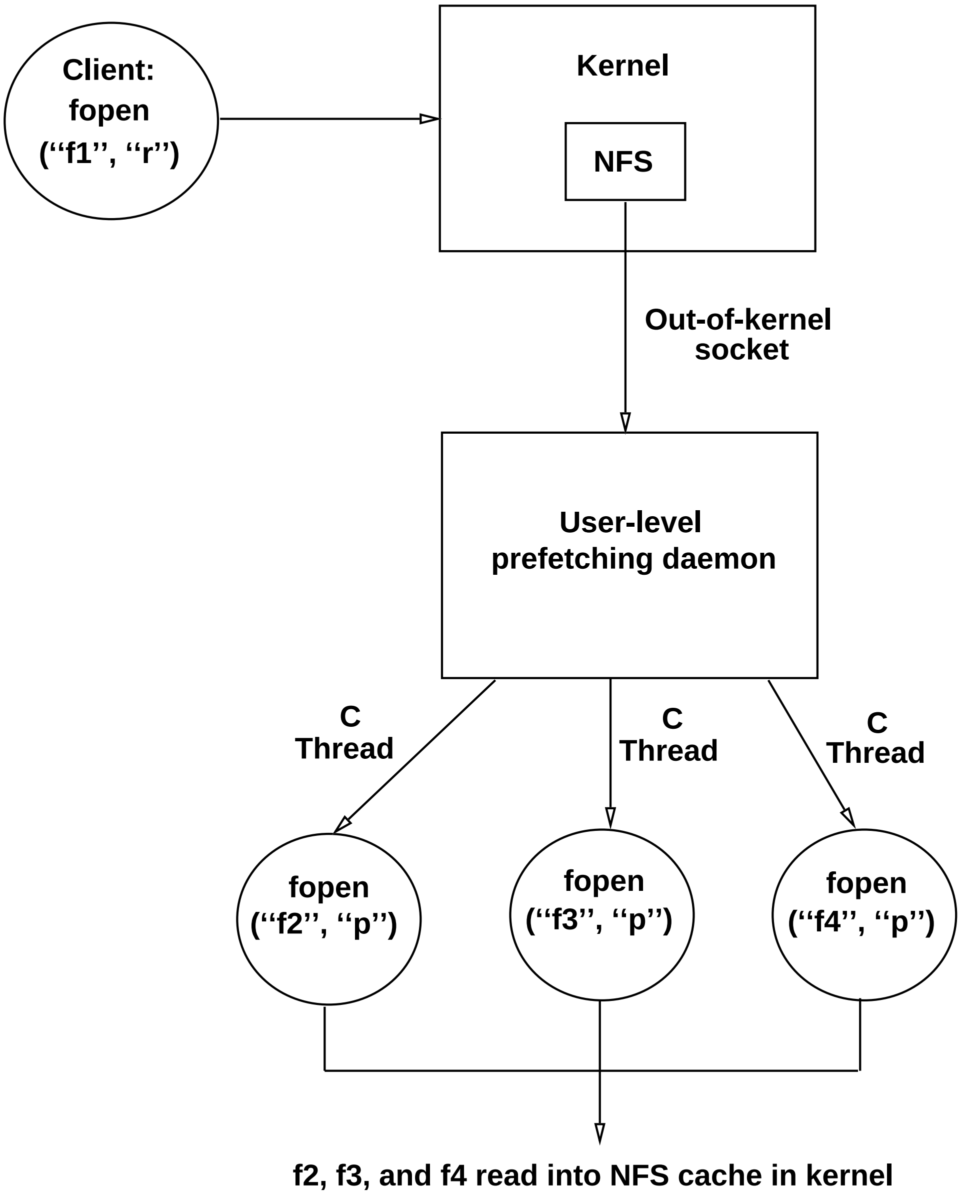

Figure 4.5: File Prefetching Implementation

user buffer. We want to skip the unnecessary copies above the NFS buffer level.

The solution: we define a special open mode "p". When the prefetching code uses `fopen("p")`, NFS knows not to send any file reference messages to the prefetching daemon. Intermediate buffer copies are also suppressed.

Filename resolution poses an additional problem. In order to resolve relative pathnames correctly as the user moves from one working directory to another, the prefetcher must be informed whenever a `chdir()` system call is executed. Furthermore, directory information must be maintained on a per-process basis: each process may have a different working directory. Given this information, name resolution becomes straightforward. For example, if process 1234 opens file `mydata`, the prefetcher looks up pid 1234's current working directory in a hash table, and prepends this string to the relative pathname provided. The final result is a fully-expanded pathname such as `/u/tait/mydata`.

The implementation is surprisingly complex, and will be described in detail in the following subsections. First, however, a quick overview of NFS is in order, since the prefetching implementation is necessarily tailored to the way NFS manipulates data.

4.4.2 NFS Overview

The first NFS data structure to consider is the *vnode*, which is the NFS equivalent of a UNIX inode. It is not necessary to examine the structure of a vnode in detail, other than to note that it encapsulates information about the remote filesystem where a file is located. However, it is critical to realize that NFS converts character-string filenames into vnodes at the time the file is opened, and then *discards* the string name. This means that it is very difficult (if not impossible) to trace any operations performed on a specific file after the initial `open()` call. Furthermore, it means that the window in which we can install our prefetching hooks is narrow: we must wait until we know that the `open()` or `exec()` has succeeded, but we must not wait so long that the character-string filename has been discarded.

An unusual feature of NFS is that its top-level vnode interface is generic: theoretically, any network file system could be installed to handle operations on vnodes. For this reason, calls to NFS through the vnode interface pass through an intricate series of macros designed to map onto calls to the filesystem of choice — in this case, NFS.

Tracing the flow of control through a `read()` operation will help make this clear. `vfs/vfs_vnode.c` contains top-level code for operations on vnodes, and the function `vn_rdwr()` is responsible for handling reads and writes. From here, the `VOP_RDWR()` macro ("vnode operation: read/write") in `vfs/vnode.h` maps us onto the NFS-specific `nfs_rdwr()` function in `nfs/nfs_vnodeops.c`. This file contains the NFS implementations of the operations that one may perform on vnodes. After checking permissions and file attributes, control is passed to another function in the same file, `rwvp()`. This is the workhorse function that actually initiates block I/O on the file in question.

The functions that perform block I/O, `bread()` and `breada()`, are found in `vfs/vfs_bio.c`. The only difference between these functions is that `breada()` starts an asynchronous read-ahead on the next block in the file. When `rwvp()` detects that blocks in a file are being read sequentially, it calls `breada()` instead of `bread()` in an effort to reduce cache misses. This is a form of prefetching *within* files, as opposed to our method, which operates *across* files.

The twisty path that NFS follows below the block I/O level need not concern us here, but it is necessary to consider one more data structure: the *uio*. This is the structure that holds most of the parameters for data transfer involving a specific vnode: buffer address, file offset, and byte count (read or write is specified in a separate parameter). Here is the format of a uio, along with its associated data structure, the *iovec*:

```
struct iovec {
        caddr_t iov_base;    /* base address of I/O buffer */
        int     iov_len;     /* count of bytes to transfer */
};
```

```
struct uio {
        struct  iovec *uio_iov;  /* ptr to above structure */
        int     uio_iovcnt;  /* number of iovecs -- typically 1 */
        off_t   uio_offset;  /* offset within file for I/O */
        int     uio_segflg;  /* segment of memory: user or system */
        int     uio_resid;   /* bytes remaining to be transferred */
};
```

The important field for prefetching purposes is `uio_segflg`, which is normally set to UIO_USERSPACE or UIO_SYSSPACE to specify a buffer in user or system memory, respectively. When prefetching a file, we would prefer not to specify an I/O buffer at all — we only want the file to be read into the NFS buffer of disk blocks, not to be copied into user space. We get around this problem by defining a new segment flag specifically for prefetching: UIO_NOSEG. We then alter NFS to skip buffer copies whenever `uio_segflg` is UIO_NOSEG. This and all other details of the prefetching implementation will be discussed in the following sections.

4.4.3 New Files

The main prefetching module is **tree.c**. It saves trees of files built during a program's execution, compares current activity with saved trees, and prefetches a tree's files if a match appears likely. **tree.c** contains the following top-level routines:

- prefetch_init(bufpages, page_size):
  Initialization code. (bufpages * page_size) is the total size (in bytes) of the NFS cache.

- prefetch_forkprog(parent, child):
  Called when a parent process forks a child.

- prefetch_execfile(pid, filename):
  Called when a program is executed under a given pid.

- prefetch_fileref(pid, filename):
  Called when a file is opened for reading. (We no longer trace creat and mkdir; they don't help with prefetching.)

- prefetch_chdir(pid, dirname):
  Called when a process changes the current working directory. This knowledge is needed for the prefetcher to resolve names properly, as described in Section 4.4.1.

- prefetch_cache_info(Lhit, Lmiss, Rhit, Rmiss): Gives tree.c a count of hits and misses in the NFS block cache, broken down into counts for local and remote files.

The prefetching daemon consists of **tree.c** and the following source files:

- **q2.c, q2.h:** A general-purpose queue management package.

- **strpool2.c, strpool2.h, defs2.h:** A substantially modified version of generic hash table routines written at Brown University. These routines allow key strings to be associated with pointers to arbitrary types.

- **socko.c, socko.h:** User-side code for the out-of-kernel socket (same modules that were used in the file system prototype).

The following files are linked into the modified kernel:

- **ksocko.c, ksocko.h:** Kernel-side code for the out-of-kernel socket (same modules that were used in the file system prototype).

- **pf_port.h:** Port assignment for prefetching (used on both sides of the socket).

- **pf.c, pf.h:** Kernel-side code to push NFS data into the socket. Also includes code to start up and shut down a socket.

4.4.4 Modifications to Existing Files

A number of kernel, NFS, and library files had to be modified. The modified library code is linked only with the prefetching daemon; all other programs continue to use the standard library.

Initialization — bsd/init_main.c: At the end of `setup_main()` (the general initialization routine for the whole kernel), call `prefetch_init()` just before returning:

```
prefetch_init(bufpages, page_size);
```

fork — bsd/kern_fork.c: In `newproc()` (the procedure responsible for forking off new processes), call `prefetch_forkprog()` with the parent and child pids:

```
prefetch_forkprog((int) rip->p_pid, (int) rpp->p_pid);
```

exec and execve — bsd/kern_exec.c: Call `prefetch_execfile()` after the file is opened in `execve()` and the credentials have been verified — just before the header is read to get the magic number. We must not wait any longer than this, or we would have to make a copy of the filename since it's overwritten by the exec. The parameters we pass out of the kernel are the pid and the filename:

```
prefetch_execfile((int) u.u_procp->p_pid, uap->fname);
```

An unpleasant note. During script processing, `execve()` jumps back 200 lines with a `goto` when it finds a shell within the script to execute. Since we do not want to call `prefetch_execfile()` again (this would overwrite the pid/program information from the previous call), we must use a local Boolean variable to prevent the call from being made more than once.

open — vfs/vfs_vnode.c: In `vn_open()` — the top-level routine for opening a file and associating it with a vnode — call `prefetch_fileref()` just before returning. The values passed out of the kernel are the pid and the filename:

```
                                        /* Call out only if: */
if ((!error) &&                         /* successful open */
    (!(filemode & FPREFETCH)) &&        /* not from the prefetcher */
    (filemode & FREAD) &&               /* opened for reading */
    ((vp->v_mode & VFMT) == VREG))      /* regular file */
  prefetch_fileref((int) u.u_procp->p_pid, pnamep);
```

chdir — vfs/vfs_syscalls.c: At the end of `chdirec()` (which handles the `chdir()` system call), call `prefetch_chdir()` just before returning. The arguments are the pid executing the `chdir()`, and the new working directory.

```
if (!error)
  prefetch_chdir((int) u.u_procp->p_pid, dirnamep);
```

Tracing Cache Hits — sys/buf.h, vfs/vfs_bio.c, bsd/init_main.c

If we were to naively call out from the block I/O functions `bread()` and `breada()` with block hit/miss information, we would inadvertently be tracing hits and misses incurred by the prefetching code itself. This is clearly undesirable: we don't care how often prefetching requests hit in the cache; we are concerned only with how well the prefetcher helps other processes avoid cache misses. We therefore do the following (which filters out system-internal reads as well as prefetching reads):

1. Rename `bread` and `breada` to `PF_bread` and `PF_breada`. Each function has one extra parameter at the end: the segment flag of the caller. As described in Section 4.4.2, the segment flag indicates whether the data is to be copied into user space, system space, or — in the case of prefetching — not copied out of the NFS buffer at all (UIO_NOSEG).

2. Define `bread` and `breada` as macros in sys/buf.h that map existing calls onto the corresponding `PF_bread` and `PF_breada` calls:

```
#define bread(V, B, S) \
   (PF_bread(V, B, S, \
```

```
((uio == NULL) ? UIO_NOSEG : \
 uio->uio_segflg)))
```

3. Define a global uio structure in bsd/init_main.c that the macro will use if there is no local uio available: `struct uio *uio = NULL;`

4. Replace the `bread` calls in `PF_breada` with calls to `PF_bread`. We don't want to use the macros in this case.

5. During execution, if `PF_bread` and `PF_breada` find that this is not a prefetching operation (segment flag is not UIO_NOSEG), and the file mode is either VREG (remote regular file) or 0 (local regular file), call `prefetch_cache_info()` with hit/miss data and the mode.

6. Code in pf.c periodically sends the accumulated hit and miss counts out to the prefetcher.

Suppressing Unnecessary Buffer Copies

Library files (linked in *only* with the prefetch code):

- fopen.c:
  Add code to support open mode "p" and modify code to drop the unnecessary lock that prevents simultaneous operations.

- fread.c:
  If prefetching, suppress copy from file buffer into user buffer.

- filbuf.c:
  Disable virtual memory mapping: `#undef _FMAP`. If prefetching, don't allocate file buffer.

- stdio.h (local version):
  Define _IOPFCH for use in all of the above files.

Kernel side:

- sys/file.h:
  Define prefetching bits FPREFETCH and O_PREFETCH; modify FMASK to preserve the FPREFETCH bit for later `read()` operations.

- sys/uio.h:
  Define UIO_NOSEG.

- bsd/sys_generic.c:
  In `rwuio()` — the function that plugs the proper values into a uio structure prior to data movement — set the segment flag to UIO_NOSEG if the file is being prefetched:

```
if (fp->f_flag & FPREFETCH)
  uio->uio_segflg = UIO_NOSEG;
else
  uio->uio_segflg = UIO_USERSPACE;
```

- bsd/kern_subr.c:
  In `uiomove()` — the function that moves data to and from NFS buffers based on parameters in a uio structure — add `case UIO_NOSEG: break;` to the case statement. This causes NFS to skip the buffer copy out of the kernel when a file is being prefetched.

4.4.5 Differences between the Simulation and the Implementation

The prefetching algorithm itself has been changed in a number of ways:

1. Cycle trees have been eliminated because the information they generated was largely redundant. In the trace-driven simulation, this was not a problem because there was no cost associated with prefetching a file that had already been brought into the cache by another part of the algorithm. In the implementation, however, redundant kernel calls have a negative impact on performance.

2. The "stupid pair" method has also been eliminated for the same reason. Here, the redundancy was even more noticeable.

3. If there is only one saved tree for a program, the implementation prefetches it immediately. This helps in the case of a program that is executed many times under identical circumstances.

4. When prefetching a tree, all files up to the one that triggered the tree selection are skipped over; we have missed our chance to prefetch any earlier files. We also skip one file *after* the trigger point because it is likely that we are already too late to prefetch it — the client process has continued to run while we were making our prefetch decision. By skipping a file, we let the prefetcher get ahead of the client, and increase the likelihood that we will be able to prefetch files before they are actually needed.

   This technique is effective primarily when we expect rapid-fire file accesses. It may turn out that the file we skip over *could* have been prefetched in time to save a cache miss, but there is no way to determine this in advance. We are willing to accept the trade-off: sacrificing a single file makes it more likely that we will be able to prefetch the remainder of the tree in time to reduce future cache misses.

5. MAXLINK has been increased to 500. The benchmarks described below consistently used trees containing more than 250 files. (Links are always allocated dynamically; MAXLINK is simply an upper bound on the number of links we allow to emanate from a single graph node.)

6. We make no attempt to prefetch directories. Remote directories cannot be cached, and local ones are usually cached, anyway.

Due to limitations of the trace data, the simulation counted files rather than bytes when computing tree sizes. The implementation, of course, computes sizes in bytes, and this required that the tree-splitting code be rewritten.

Cache management is handled entirely by NFS on an LRU basis. The implementation, knowing the size of the cache, merely chooses which files to prefetch and when to prefetch them. In addition, NFS traces cache hits and misses on a block-by-block basis, so our hit/miss information is now at the granularity of blocks rather than whole files — a desirable attribute.

4.4.6 Evaluation

We have gathered results from two benchmarks to evaluate the prefetching implementation. The first test models the sort of file access patterns one might observe in a wide range of applications. The second benchmark is a demanding task — a kernel build — chosen specifically because one would *not* expect prefetching to do particularly well in this case. Since we are particularly interested in how well our method is suited to a wireless mobile computing environment, we tested the benchmarks using both fast hardwired and slow wireless connections from the client to the file server.

Benchmark 1: Synthetic Read/Sleep Loop

Our first test involved reading several hundred files in a fixed order, sleeping for a certain amount of time after each read. A delay time of zero between reads is especially challenging: the client never stops reading files, so prefetching decisions must be made promptly and accurately. Non-zero delay times reflect the sort of behavior found in programs such as text processors and compilers, which spend some time processing each file after reading it. Tests involving non-zero delay times show prefetching in the best light: there is ample time available both for computation and for reading files ahead of the client.

Table 4.6 shows the results when the benchmark is run on a 486 processor with 2.7MB of its 16MB of memory dedicated to the NFS block cache. The client is connected to the file server by a hardwired 10 megabits/second Ethernet connection. Table 4.7 shows the same data for a portable computer with a 386 processor (about 2.5 times slower than the 486), 8MB of memory, and a cache size of 808KB, connected to the server by a wireless 2 megabits/second NCR WaveLAN radio link. In both cases, the file server is a Sun SparcStation-2. NFS block size throughout is 8KB.

In the tables, the "Time" columns report the measured time of the benchmarks, while the "File Time" columns show how much time was actually spent in processing files: the total delay time has been subtracted out. The "Misses" columns report the number of cache misses: no hits or misses are filtered, as some were in the simulation. "PF" and "No PF" refer to tests with and without prefetching, respectively.

For both connection speeds, prefetching substantially reduces both the number of cache misses and the time consumed in file processing in almost all instances. The one exception is the 10 Mb/sec case with no delay between reads: both the benchmark and the network are running so fast that prefetching simply gets in the way by stealing processor cycles from the benchmark.

Although the percentage by which prefetching reduces execution time is less for the 2 Mb/sec connection than the 10 Mb/sec, two points should be emphasized. First, from a client's standpoint, reducing the length of a long job by 40 or 50 percent may well be more valuable than cutting 60 or 70 percent off the time of a short job. Clients with the slower link will actually do better than those with the faster link in terms of real time saved. Second, prefetching improves performance all the way down to a delay time of zero in the 2 Mb/sec case. (Because the benchmark must wait longer for files to arrive, there are plenty of excess CPU cycles available for use by the prefetching code. This would become an even greater advantage if the processor were a 486 rather than a 386.) Success even with a delay time of zero in the 2Mb/sec case means that prefetching is applicable in a wider range of circumstances when using slow links than when using fast ones. All of this bodes well for mobile clients using slow wireless links.

Since our prefetching scheme is dependent on the availability of client CPU cycles, we ran one more test in which a (relatively) slow 386 CPU was paired with a fast 10 Mb/sec connection. The results are shown in Table 4.8. When there is no delay between reads, CPU cycles become a critical resource: prefetching does

Delay	Misses No PF	Misses PF	Time No PF	Time PF	File Time No PF	File Time PF	Percent Reduction
0.0	397	279	12.8	13.8	12.8	13.8	-7.8
0.1	397	25	48.2	40.7	11.4	3.9	65.8
0.5	397	5	195.6	187.2	11.6	3.2	72.4
1.0	397	5	380.5	373.5	12.5	5.5	56.0

Table 4.6: Read/Sleep Benchmark (10 Mbits/sec Connection, 486 CPU)

431 block accesses (369 files, 1281828 bytes).

All times are measured in seconds.

"File Time" omits time spent sleeping.

"Percent Reduction" refers to file time, not total time.

Delay	Misses No PF	Misses PF	Time No PF	Time PF	File Time No PF	File Time PF	Percent Reduction
0.0	397	146	80.0	55.8	80.0	55.8	30.3
0.1	397	60	133.5	91.3	96.7	54.5	43.6
0.5	397	21	275.1	230.4	91.1	46.4	49.1
1.0	397	18	470.6	416.8	102.6	48.8	52.4

Table 4.7: Read/Sleep Benchmark (2 Mbits/sec Connection, 386 CPU)

not improve the miss rate at all, and the benchmark's execution time is more than *doubled* due to contention for processor cycles. When the delay time is nonzero, excess CPU cycles become available, and prefetching is able to improve performance.

Benchmark 2: Kernel Build

During the implementation of the prefetching code, it was often necessary to make changes to one kernel source file, and then wait patiently while the `make` program figured out which file to recompile, compiled it, and then re-linked the entire kernel. This build procedure accessed hundreds of files in a patterned way,

Delay	Misses No PF	Misses PF	Time No PF	Time PF	File Time No PF	File Time PF	Percent Reduction
0.0	397	397	17.8	36.7	17.8	36.7	-106.2
0.1	397	61	54.6	48.2	17.8	11.4	36.0
0.5	397	15	202.8	192.5	18.8	8.5	54.8
1.0	397	11	388.5	378.0	20.5	10.0	51.2

Table 4.8: Read/Sleep Benchmark (10 Mbits/sec Connection, 386 CPU)

and thus might seem to be a good candidate for improvement by prefetching. For several reasons, however, this is not the case.

First, much time is spent in calling `stat()` on hundreds of remote source files. Our prefetching algorithm can provide no assistance here because we never prefetch a file unless it has been opened or executed. Second, even if the build required multiple source files to be recompiled, prefetching would be unlikely to help. Most of the compiler and common `.h` files fit comfortably in the cache, and each source file is compiled only once. Therefore, the prefetcher would usually be requesting files that were already cached or source files that were no longer needed.

Finally, linking a large number of object files at the end of the build is roughly equivalent to the read/sleep benchmark described above with a delay time of zero: the linker requires near-zero processing time on each file after reading it in. As shown above, a delay time of zero produces much the least impressive benefits from prefetching.

A kernel build is thus a particularly grueling test of the prefetching implementation. Table 4.9 shows the results of multiple trials on a 486 processor with a 10 Mb/sec hardwired connection, a 386 with a 10 Mb/sec hardwired connection, and a 386 with a 2 Mb/sec wireless connection. Using the faster connection, the results were about as expected: prefetching helped only slightly, if at all — it actually *increased* the execution time by a small amount in Trial 3 using a 486 CPU, and in all cases using a 386. When using a slow wireless connection, however, cache misses are considerably more costly, and prefetching was able to improve the glacially slow

Connection and CPU	Trial	Misses No PF	Misses PF	Time No PF	Time PF	Percent Reduction
10 Mbits/sec 486 CPU	1	732	657	194.0	192.1	1.0
	2	691	632	192.7	185.8	3.6
	3	799	615	187.6	187.9	-0.2
10 Mbits/sec 386 CPU	1	1081	977	272.0	278.6	-2.4
	2	1082	973	270.2	282.3	-4.5
	3	1081	970	270.0	276.6	-2.4
2 Mbits/sec 386 CPU	1	1083	1015	1096.4	1032.9	5.8
	2	1085	1050	1091.3	1032.7	5.4
	3	1082	1070	1159.5	1107.1	4.5

Table 4.9: Kernel Build Benchmark

The build involved 3456 block accesses.

Time is measured in seconds.

speed of a kernel build by about one minute.

It is reassuring to know that, even in this inimical test, prefetching does not make things worse. It is even more encouraging to observe that prefetching actually helps in the 2 Mb/sec wireless case — another piece of evidence that our method is particularly well-suited to mobile computing.

4.4.7 Discussion

The most common prefetching data structure occupies about 55 bytes, and thousands of these can be created during long runs. In our trace-driven simulation, garbage collection had the effect of inducing an equilibrium on the number of nodes in existence, with the result that total program size remained relatively constant around 300KB. (Smaller caches required more program space than large ones due to the greater number of trees that had to be split.) In the implementation, space requirements have been reduced to about 150KB thanks to compression and re-implementation of critical data structures.

The prefetch recommendations produced by analysis of the working forest can and should be treated as a hint. The cache management policy remains LRU. Either the working forest or the stacks of saved trees could at any time be reduced or thrown away, with the only negative effect being a reduced hit rate, bounded below by the hit rate that LRU would produce by itself. Thus, the algorithm is not sensitive to any particular choices for the size of data structures, and performance and resource use can be traded off against each other as necessary. Nevertheless, it is desirable to maintain as much of this information for as long as possible. To avoid incurring the "learning curve" costs after every new login or reboot, the data structures could be checkpointed to disk during idle periods.

A disadvantage of our algorithm is that it requires information from two parts of an operating system that are increasingly being separated: process management and the file system. Fortunately, the information needed by the cache manager is minimal: the user ID of any individual whose file access behavior is being "snooped," and notice of every `fork` and `exec` call made by any of that individual's processes. Since our algorithm only provides hints, this information could be batched and/or provided late to the file system, with reduced performance being the only cost. The other information needed to construct trees — namely the user ID, and process ID of each `open`, `creat`, or `mkdir` — is typically available within a file system implementation, where it is needed to check permissions and establish open file tables.

4.5 Related Work

File prefetching is not a new idea. Alonso's group has contemplated a number of stashing methods for inclusion in their FACE file system [2, 3]. In order of increasing sophistication, they are:

1. Make users responsible for which files are stashed. Various methods are proposed. One is enumerating the files that are to be stashed in a ".stashrc" file. Another is making an explicit "stash" command available to the user. The last is to have the user tell the system to record and stash all files used between

two selected points in time.

2. Have each application stash what it needs.

3. Engineer the file system to read and understand certain key files that indicate which other files are needed to perform a task; e.g., "'makefiles."

4. Have the system stash files used in the last several commands given by a user.

The FACE report does not explain how these ideas might be achieved.

A stash mechanism that depends on users for correct and timely information or action is unsatisfactory, for two reasons. First, many users will refuse to do the necessary work required to keep a stash primed with the correct contents. Second, users will be unable to fully describe what should be in the stash. In some cases users will forget that certain input files are required; more often, users will be ignorant of certain files required by the *implementation* of an application, which is necessarily unknown to them. It is the task of modern file systems to remove this sort of burden from the user.

To support disconnected operation in the Coda file system [51], Kistler and Satyanarayanan have implemented a technique known as *hoarding* [23]. This is a cache management scheme designed to increase the likelihood that a user will be able to continue working during periods of total disconnection from file servers. Hoarding combines traditional cache management based on recency with a *hoard database* (HDB) that is similar to the ".stashrc" file proposed for use in the FACE system. The HDB contains a prioritized list of files that should be kept in the local cache. To keep this list as simple and as terse as possible, entire directories (with all descendants, if desired) can be lumped together as a single entry in the HDB. The HDB can be constructed either explicitly, or by having the system observe file accesses during a fixed interval. Periodic *hoard walking* ensures that the user's cache conforms to the constraints specified in the HDB.

When users cooperate, hoarding works quite well, allowing the file system to prefetch files or to bump them from the cache as needed. However, relying on information provided by users can lead to all of the attendant problems described above.

Another research effort with prefetching as its goal is the Transparent Informed Prefetching (TIP) work described in [43]. The TIP designers' outlook is more broad than ours, as they consider prefetching from devices as well as from file servers. However, the basic goal of initiating I/O in advance of need remains constant. The central aim of TIP is to design an interface to a prefetching engine so that higher levels can pass down prefetching hints in a fashion that is at once efficient and modular. The TIP approach possesses an advantage over ours in that prefetching is driven not by deductions made after snooping, but rather by certain knowledge provided in advance by higher levels. There is therefore no danger that disastrously incorrect prefetch decisions might trash the cache. On the other hand, TIP must act within the interval between when the higher level learns of the need to do I/O and when it actually initiates I/O; it remains to be seen how long this interval typically is. Data in [43] demonstrate performance improvements up to 30%. A further advantage of TIP is that it can help applications that make "random" accesses.

The work of Curewitz, Krishnan, and Vitter [11] takes an unusual approach to prefetching by using data compression techniques to predict future accesses. A compression program must detect patterns — and their associated probabilities of occurrence — in the data stream in order to compress data effectively; prefetching based on this approach makes probabilistic predictions based on these patterns. The techniques proposed by Curewitz et al. for *practical* prefetching are deliberately simple: one of them, in fact, is a generalization of the "smart pair" and trigram schemes that were used as a basis for comparison with our tree method in Section 4.3.

It is unclear how well data-compression-based prefetching would work in the "engineering/office" environment [38] for which our own prefetching technique is targeted. The trace-driven simulations of Curewitz et al. are based on CAD and database traces; the focus of the work is *prepaging* rather than prefetching (so decisions are often more time-critical); and the cache sizes in the simulations are extremely small: only 10 pages, with a brief comment that a (still small) cache of 50 pages produces similar results.

The work most closely related to our own is Korner's work on detecting and exploiting the *block access patterns of individual files* [25]. Several important aspects make Korner's work different. First, the detection of patterns is an off-line process performed by two expert systems, whereas our method provides on-line "real-time" detection and exploitation. Further, whereas our method is an adjustment to LRU, many of the access patterns Korner discusses (e.g., MRU) are alternatives to LRU; thus, a cache manager would require substantial re-coding to benefit from the noted access patterns. Finally, Korner's objective is the more limited problem of detecting and exploiting the block access patterns within a file, not file access patterns within an application. Since a large proportion of files are accessed in their entirety [39], it is not clear what percentage of files could benefit from Korner's prefetch rules. Our method benefits any application whose file access behavior is predictable. In addition, Korner's proposed exploitation is for the file server's in-memory cache of disk blocks, not for a client-side cache of server contents.

Korner's work can complement ours in two ways. One would be to use our file prefetching method in the client (file) cache while using Korner's technique for the server-side (block) cache. Better yet would be to couple the techniques for use in "remote open" file systems that move blocks, and not files, between client and server. In this case, a client-side block cache would use our technique to determine which files should have blocks in the cache, and Korner's technique would pick the right blocks of those files.

Similar to Korner's work is that of Kotz and Ellis (see, for example, [26]) on the uses of prefetching to increase I/O bandwidth in MIMD shared-memory multiprocessors. Their work also focuses on prefetching blocks within files. They exploit the fact that many scientific and database applications running on multiprocessors exhibit simple and well-known patterns of sequential access coupled with read-only or write-only behavior: e.g., read every Nth block of the entire file. Their prefetching methods are geared to these access patterns.

Our prefetching scheme has two decided advantages over any of these other methods. First, it is more on-line than any of them: there is no off-line computation or periodic analysis required. Second, and more importantly, our prefetching method is completely transparent to clients. There need be no special stash information files or modifications to existing programs, and this is a compelling advantage.

4.6 Summary

We have presented an algorithm that detects and exploits file working sets. It is used to add prefetching intelligence to the basic LRU cache management strategy. The technique is applicable to any UNIX-style system, is independent of the cache consistency algorithm, imposes little overhead, can be added to existing standard software, and is effective in improving the client cache miss rate that would be obtained by LRU alone. Decreasing cache miss rate speeds applications and renders clients more independent of file server loss.

The algorithm's trait of spending client cycles (and, to a lesser extent, client memory) in return for more effective use of client cache space and fewer on-demand network operations leads us to believe it would be most effective in an environment of radio-connected "notebook" workstations. Our algorithm seems to offer especially impressive advantages in these circumstances.

5

Conclusion

5.1 Contributions

The purpose of our research has been to show that client mobility can be supported without severely degrading performance or availability, and without unnecessarily complicating a user's view of the system. We have developed two major ideas to realize our goals.

Our first idea, efficient variable-consistency replication, provides an unprecedented degree of decoupling between clients and servers. This is a particularly valuable attribute for mobile clients and the servers they employ: a client-server relationship may be terminated by either party, at any time, for any reason. In order to cope with the increased difficulty of finding the most up-to-date version of a file in such circumstances, we allow each client to specify the level of consistency required on a per-read basis. Furthermore, only clients who insist on high consistency are required to pay the performance cost; we do not amortize such costs over all clients. Finally, even clients who demand strict consistency will usually receive excellent performance: only the *initial* strict read operation requires synchronously contacting multiple servers and clients.

The primary benefit of our second idea, intelligent file prefetching, is improved performance due to a reduced number of cache misses. A secondary benefit is increased availability of files during disconnection from the server, thanks to a cache that has been intelligently filled with prefetched files. Both of these benefits are of particular importance when the client-server link is slow, unreliable, or both — as is the case with wireless mobile clients. By distributing file accesses over time, we reduce the "burst" load on the network, and increase the effective bandwidth perceived by the client. The greater the ratio of CPU speed to network speed, the greater the benefits enjoyed by the client during prefetching.

5.2 Future Work

5.2.1 General

The numerous parameters on which our work relies need to be investigated more rigorously. Some of these parameters, such as the interval between pickups, are intended to be dynamic, and measurements taken on a fully functional implementation should provide insight into how to adjust these parameters as the system is running. In other cases — such as the percentage of overlap between trees that defines a "successful" prefetch — it is not yet clear whether a dynamic or static value is preferable. We have already run enough experiments to determine that a wide range of values may be used successfully for most of the critical parameters (see, for example, the pickup interval experiments in Section 3.4). Ultimately, however, it will be important to do further research that will establish firmer boundaries on what constitute "acceptable" and "optimal" values for our parameters.

5.2.2 Efficient Variable-Consistency Replication

The next step, of course, is to implement the full prototype design described in Section 3.5, and this project is already underway. In addition to working out the lowest-level details of the algorithms, the full prototype will act as a tool for evaluating the usability of the dual-read-call interface. The current version is too crude to provide any help in this area; in the final prototype, it will be advisable to use a modified shell that automatically converts filenames into the rigid format required by the prototype.

Measuring the scalability of our file service will be another challenge. One technique that might be useful in this area is *isoefficiency* analysis [28]: a scalability metric that relates problem size to the processing power required to ensure a proportional rate of speedup. To improve scalability, we may consider a scheme such as that used in QuickSilver's OBJ interface [56], which packs as much information as possible into each network message, thus reducing the total number of messages.

Many other issues remain to be considered. Authentication and security, for example, have not been dealt with except to provide clean boundaries for their enforcement. We deliberately made naming an orthogonal issue in our system; it will ultimately be necessary to choose a specific scheme for name resolution. Finally, in order to provide a generally usable file service on which we can make realistic performance measurements, a full-scale implementation will be required beyond the prototype.

5.2.3 Intelligent File Prefetching

We see many directions for future work, some to fix shortcomings noted earlier, and some to extend the work.

An obvious step to improve our work is to incorporate the prefetching logic within the file system rather than encapsulating the logic in an external process. We originally conceived an implementation in the file system but were stopped by the difficulty of multi-thread programming within the kernel. Kernelized systems that have the file system in a process are becoming common and are the obvious solution.

A presumed beneficial side effect of implementing in the file system is that we would not be reluctant to use more information about file accesses in order to make our algorithm smarter. Our experience suggests at least three ways in which extra information might reduce waste during prefetching:

1. Don't suffer the overhead of "prefetching" files known already to be in the cache.

2. Detect a tight I/O loop (such as the read/sleep tests with delay of zero) and turn off prefetching during these times.

3. Detect bursts of activity within directories and fetch the entire directory.

Another benefit of implementing within the file system is the ability to quantify the costs of prefetching. Currently, our simulations and implementation benchmarks only provide us with an end-to-end check on costs: whatever our technique costs in terms of client CPU cycles, network bandwidth, and server load is more than paid for by the benefits of prefetching, given a sufficient number of excess client cycles. Even with the prefetcher within the file system, however, quantifying costs is non-trivial: it is possible to make only "correct" prefetching decisions and still have a negative overall effect because the files were prefetched too late to be of use to the client. Furthermore, there is another case in which prefetching initially appears to hurt, but actually helps. Suppose there are no excess CPU cycles, but the prefetcher steals some, anyway. Because cache misses can be quite expensive, the stolen cycles may end up paying for themselves many times over if they save a cache miss.

As described, when the prefetcher shuts down, its knowledge is lost. It might be useful to be able to store saved trees in files, so that information used for prefetching could persist for an extended time. Another possibly desirable extension would be to provide the client with an interface to the server such that prefetching could be done more efficiently. A special "batch" interface might allow the server to schedule its activity more efficiently.

Our prefetching technique depends on repeated executions of application programs exhibiting similar file reference behavior. We suppose that this condition is more likely to be satisfied on a personal workstation than on a timeshared machine, and that is why we have aimed our current effort toward the case wherein the client is a single-user workstation. It is an open question how much our technique would help or hurt the file system of a many-user machine.

A more substantial concern for the future is that the algorithm as it now stands has a limited definition of what constitutes a "computation" — namely, a tree of forked processes. The strong trend in operating systems is toward providing a multiplicity of ways to spread a computation across processes: message passing between clients and servers, shared memory, multiple threads of control within a process, and so on. As this trend continues, it will become harder to track all the locations and actions of a computation. This uncertainty will confound our method. Consider the difficulty of tracing the file accesses of a computation that is accomplished partly by calling a multi-threaded server on another machine. We expect the tracking and control of distributed computations (i.e., the distributed computing equivalent of "network management") to be an area of research in the near future.

One might also consider improving the algorithm by taking process lifetimes into account. If a process is very short-lived, it is unlikely that any on-line prefetching scheme will be of help: by the time we figure out what to prefetch, the process may well have terminated. In Cabrera's study [8], more than 78% of processes (up to 95% in some cases) had lifetimes of less than one second. By focusing our prefetching efforts on the relatively small percentage of longer-lived processes, our chances for success are improved — particularly if a significant percentage of total file accesses are made by the longer-lived processes.

It is conceivable that our algorithm could be improved by using a representation other than trees for file working sets. Some form of regular expressions, for example, might be made to work. However, it is not clear that the approximate matching we currently use for trees would be well-suited to regular expressions. Furthermore, the complexities involved in splitting trees have no obvious analogue in regular expressions.

Finally, the work on informed prefetching [43], when contrasted with ours, poses an open question: what are the limits and possibilities of "transparent" and "informed" file prefetching, and in which environments is one technique the more desirable?

5.3 Summary

This dissertation has described two ideas designed to support mobile file system clients: efficient variable-consistency replication and intelligent file prefetching. Algorithms, implementations, and analyses have been presented for both of these ideas. Although both concepts have general applicability outside the scope of mobile computing, each idea is particularly well-suited to the task of supporting mobile clients.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young.
Mach: A New Kernel Foundation for UNIX Development.
In *Proc. 1986 USENIX Summer Conf.*, pages 93–112, June 1986.

[2] R. Alonso, D. Barbara, and L. L. Cova.
Augmenting Availability in Distributed File Systems.
Technical Report CS-TR-234-89, Princeton University, October 1989.

[3] R. Alonso, D. Barbara, and L. L. Cova.
FACE: Enhancing Distributed File Systems for Autonomous Computing Environments.
Technical Report CS-TR-214-89, Princeton University, March 1989.

[4] P. A. Alsberg and J. D. Day.
A Principle for Resilient Sharing of Distributed Resources.
In *Proc. Second Intl. Conf. on Software Engineering*, pages 562–570, October 1976.

[5] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout.
Measurements of a Distributed File System.
In *Proc. Thirteenth Symp. on Operating System Principles*, pages 198–212. ACM, October 1991.

[6] A. Bhide, E. N. Elnozahy, and S. P. Morgan.
A Highly Available Network File Server.
In *Proc. 1991 USENIX Winter Conf.*, January 1991.

[7] M. Blaze and R. Alonso.
Dynamic Hierarchical Caching in Large-Scale Distributed File Systems.
In *Proc. Twelfth Intl. Conf. on Distributed Computing Systems*. IEEE, June 1992.

[8] L.-F. Cabrera.
The Influence of Workload on Load Balancing Strategies.
Technical Report RJ5271, IBM Almaden Research Center, August 1986.

[9] L.-F. Cabrera and J. Wyllie.
QuickSilver Distributed File Services: An Architecture for Horizontal Growth.

In *Proc. Second IEEE Conf. on Computer Workstations*, pages 23–37, March 1988.

[10] E. C. Cooper and R. P. Draves.
C Threads.
Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988.

[11] K. M. Curewitz, P. Krishnan, and J. S. Vitter.
Practical Prefetching via Data Compression.
In *Proc. SIGMOD '93*. ACM, 1993.

[12] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan.
Making Data Structures Persistent.
Journal of Computer and System Sciences, 38(1):86–124, February 1989.

[13] C. S. Ellis and R. A. Floyd.
The Roe File System.
In *Proc. Third Symp. on Reliability in Distributed Software and Database Systems*, pages 175–181. IEEE, October 1983.

[14] R. A. Floyd and C. S. Ellis.
Directory Reference Patterns in Hierarchical File Systems.
IEEE Trans. on Knowledge and Data Engineering, 1(2):238–247, June 1989.

[15] H. Garcia-Molina.
Elections in a Distributed Computing System.
IEEE Trans. on Computers, C-31(1):48–59, January 1982.

[16] D. K. Gifford.
Weighted Voting for Replicated Data.
In *Proc. Seventh Symp. on Operating System Principles*, pages 150–162. ACM, December 1979.

[17] C. G. Gray and D. R. Cheriton.
Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency.
In *Proc. Twelfth Symp. on Operating System Principles*, pages 202–210. ACM, December 1989.

[18] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier.
Implementation of the Ficus Replicated File System.
In *Proc. 1990 USENIX Summer Conf.*, pages 63–71, June 1990.

[19] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan.
Recovery Management in QuickSilver.
ACM Trans. on Computer Systems, 6(1):82–108, February 1988.

[20] J. S. Heidemann, T. W. Page, R. G. Guy, and G. J. Popek.

Primarily Disconnected Operation: Experiences with Ficus.
In *Proc. Second Workshop on the Management of Replicated Data*, pages 2–5. IEEE, November 1992.

[21] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West.
Scale and Performance in a Distributed File System.
ACM Trans. on Computer Systems, 6(1):51–81, February 1988.

[22] J. Ioannidis, D. Duchamp, and G. Q. Maguire Jr.
IP-based Protocols for Mobile Internetworking.
In *Proc. SIGCOMM '91*, pages 235–245. ACM, September 1991.

[23] J. J. Kistler and M. Satyanarayanan.
Disconnected Operation in the Coda File System.
In *Proc. Thirteenth Symp. on Operating System Principles*, pages 213–225. ACM, October 1991.

[24] S. R. Kleiman.
Vnodes: An Architecture for Multiple File System Types in Sun UNIX.
In *Proc. 1986 USENIX Summer Conf.*, pages 238–247, June 1986.

[25] K. Korner.
Intelligent Caching for Remote File Service.
In *Proc. Tenth Intl. Conf. on Distributed Computing Systems*, pages 220–226. IEEE, May 1990.

[26] D. Kotz and C. S. Ellis.
Practical Prefetching Techniques for Parallel File Systems.
In *Proc. First Intl. Conf. on Parallel and Distributed Information Systems*, pages 182–189. IEEE, December 1991.

[27] P. Kumar.
Coping with Conflicts in an Optimistically Replicated File System.
In *Proc. First Workshop on the Management of Replicated Data*, pages 60–64. IEEE, November 1990.

[28] V. Kumar, G. Y. Ananth, and V. N. Rao.
Scalable Load Balancing Techniques for Parallel Computers.
Technical Report 91-55, University of Minnesota, November 1991.

[29] B. Liskov, R. Gruber, P. Johnson, and L. Shrira.
A Replicated UNIX File System.
Operating Systems Review, 25(1):60–64, January 1991.

[30] T. Mann, A. Hisgen, and G. Swart.
An Algorithm for Data Replication.
Technical Report 46, Digital Systems Research Center, June 1989.

[31] K. Marzullo and F. Schmuck.
Supplying High Availability with a Standard Network File System.
In *Proc. Eighth Intl. Conf. on Distributed Computing Systems*, pages 447–453. IEEE, June 1988.

[32] D. L. Mills.
Internet Time Synchronization: The Network Time Protocol.
IEEE Trans. on Communications, 39(10):1482–1493, October 1991.

[33] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith.
Andrew: A Distributed Personal Computing Environment.
Communications of the ACM, 29(3):184–201, March 1986.

[34] D. Muntz and P. Honeyman.
Multi-level Caching in Distributed File Systems.
In *Proc. 1992 USENIX Winter Conf.*, January 1992.

[35] M. N. Nelson, B. B. Welch, and J. K. Ousterhout.
Caching in the Sprite Network File System.
ACM Trans. on Computer Systems, 6(1):134–154, February 1988.

[36] B. C. Neuman.
The Virtual System Model for Large Distributed Operating Systems.
Technical Report 89-01-07, University of Washington, April 1989.

[37] B. C. Neuman.
The Prospero File System: A Global File System Based on the Virtual System Model.
In *Proc. USENIX File Systems Workshop*, pages 13–28, May 1992.

[38] J. Ousterhout and F. Douglis.
Beating the I/O Bottleneck: A Case for Log-Structured File Systems.
Operating Systems Review, 23(1):11–28, January 1989.

[39] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson.
A Trace-Driven Analysis of the UNIX 4.2 BSD File System.
In *Proc. Tenth Symp. on Operating System Principles*, pages 15–24. ACM, December 1985.

[40] T. W. Page, R. G. Guy, J. S. Heidemann, G. J. Popek, W. Mak, and D. Rothmeier.
Management of Replicated Volume Location Data in the Ficus Replicated File System.
In *Proc. 1991 USENIX Summer Conf.*, pages 17–29, June 1991.

[41] J.-F. Paris.

Voting with Witnesses: A Consistency Scheme for Replicated Files.
In *Proc. Sixth Intl. Conf. on Distributed Computing Systems*, pages 606–612. IEEE, 1986.

[42] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline.
Detection of Mutual Inconsistency in Distributed Systems.
IEEE Trans. on Software Engineering, SE-9(3):240–247, May 1983.

[43] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan.
A Status Report on Research in Transparent Informed Prefetching.
Operating Systems Review, 27(2):21–34, April 1993.

[44] G. J. Popek and B. J. Walker.
The LOCUS Distributed System Architecture.
MIT Press, 1985.

[45] E. Rahm.
Recovery Concepts for Data Sharing Systems.
In *Proc. Twenty-First Intl. Symp. on Fault-Tolerant Computing*, pages 368–375. IEEE, June 1991.

[46] K. K. Ramakrishnan, P. Biswas, and R. Karedla.
Analysis of File I/O Traces in Commercial Computing Environments.
ACM Performance Evaluation Review, 20(1):78–90, June 1992.

[47] A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, and K. Yueh.
RFS Architectural Overview.
In *Proc. 1986 USENIX Summer Conf.*, pages 248–259, June 1986.

[48] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon.
Design and Implementation of the Sun Network Filesystem.
In *Proc. 1985 USENIX Summer Conf.*, pages 119–130, June 1985.

[49] M. Satyanarayanan.
A Study of File Sizes and Functional Lifetimes.
In *Proc. Eighth Symp. on Operating System Principles*, pages 96–108. ACM, December 1981.

[50] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West.
The ITC Distributed File System: Principles and Design.
In *Proc. Tenth Symp. on Operating System Principles*, pages 35–50. ACM, December 1985.

[51] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere.
Coda: A Highly Available File System for a Distributed Workstation Environment.

IEEE Trans. on Computers, 39(4):447–459, April 1990.

[52] B. Sidebotham.
Volumes: The Andrew File System Data Structuring Primitive.
In *Proc. EUUG Conference*, Autumn 1986.

[53] A. Siegel, K. Birman, and K. Marzullo.
Deceit: A Flexible Distributed File System.
In *Proc. 1990 USENIX Summer Conf.*, pages 51–61, June 1990.

[54] C. Staelin and H. Garcia-Molina.
File System Design Using Large Memories.
Technical Report CS-TR-246-90, Princeton University, June 1990.

[55] D. B. Terry.
Caching Hints in Distributed Systems.
IEEE Trans. on Software Engineering, SE-13(1):48–54, January 1987.

[56] M. Theimer, L.-F. Cabrera, and J. Wyllie.
QuickSilver Support for Access to Data in Large, Geographically Dispersed Systems.
In *Proc. Ninth Intl. Conf. on Distributed Computing Systems*, pages 28–35. IEEE, June 1989.

[57] B. Welch and J. Ousterhout.
Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System.
In *Sixth Intl. Conf. on Distributed Computing Systems*, pages 184–189. IEEE, May 1986.