Resolving File Conflicts in the Ficus File System*

Peter Reiher, John Heidemann, David Ratner, Greg Skinner, Gerald Popek
Department of Computer Science
University of California, Los Angeles

Abstract

Ficus is a flexible replication facility with optimistic concurrency control designed to span a wide range of scales and network environments. Optimistic concurrency control provides rapid local access and high availability of files for update in the face of disconnection, at the cost of occasional conflicts that are only discovered when the system is reconnected. Ficus reliably detects all possible conflicts. Many conflicts can be automatically resolved by recognizing the file type and understanding the file's semantics. This paper describes experiences with conflicts and automatic conflict resolution in Ficus. It presents data on the frequency and character of conflicts in our environment. This paper also describes how semantically knowledgeable resolvers are designed and implemented, and discusses our experiences with their strengths and limitations. We conclude from our experience that optimistic concurrency works well in at least one realistic environment, conflicts are rare, and a large proportion of those conflicts that do occur can be automatically solved without human intervention.

1 Introduction

The value of file replication is widely recognized, but replication of updatable files leads immediately to consistency problems. File replicas can be partitioned from each other for a variety of reasons, ranging from failures of machines and networks to intentionally intermittent connections (e.g., connection via modem, or replicas on portable machines that are not always attached to a network). If no efforts are taken, partitioning can permit conflicting updates to different replicas of a file. Much of the value of replication is based on all replicas being identical, so inconsistent updates are a potentially serious problem.

Early solutions to the problem relied on various conservative algorithms that prevented conflicting updates to different replicas [1]. These solutions used a wide variety of mechanisms, but their common theme is that they refuse updates that have any possibility of causing conflicting updates. These solutions trade availability for consistency. When consistency of replicas is of vital importance, conservative solutions are preferable.

However, experience with file access by typical users has shown that many files are only accessed by a single user [10]. Of those that are shared by multiple users, few are updated by more than one user. In such environments, a mechanism that prevents one user from updating a file in favor of preserving the update ability of other users who might never generate an update is seriously flawed. Conservative replication mechanisms exhibit this flaw.

Optimistic replication mechanisms do not. They allow any replica of a file to be updated at any time. This choice ensures that users who need to update a file can do so when any replica is available. However, optimistic mechanisms gain this availability by trading off consistency. Since any replica can be updated, two non-communicating replicas can be changed independently, leading to *conflicts*, i.e., different replica contents. To maintain consistency, a system with optimistic replication must detect and recover from such conflicts.

The improved availability of optimistic systems must be weighed against the frequency and cost of recovering from conflicts. A hypothesis of this paper is that the cost of optimism is low in many environments.

To test this hypothesis, this paper reports conflict resolution experiences with Ficus, an optimistic file system developed at UCLA [4]. Ficus has supported

*This work was sponsored by the Defense Advanced Research Projects Agency under contract N00174-91-C-0107. Gerald Popek is also affiliated with Locus Computing Corporation.

The authors can be reached at 4760 Boelter Hall, UCLA, Los Angeles, CA, 90024, or by electronic mail to `ficus@cs.ucla.edu`.

the primary computing needs of over a dozen users at UCLA for more than three years.

Ficus has a general architecture for dealing with file conflicts. Conflicts are automatically detected and examined to determine if they can also be resolved automatically. Special programs called *resolvers* handle conflicts that can be dealt with automatically.

Ficus is able to resolve almost all conflicts for a particularly important class of files—directories. Ficus supports a Unix-style directory system; the semantics of Unix directories provide that almost every conflict that can occur in them can be automatically resolved. Directory conflicts could be resolved by submitting them to a resolver that implements the algorithms necessary to resolve their conflicts, but the integrity of the Unix file system is so closely linked to its directories that we have chosen to put the algorithms into Ficus itself.

We have instrumented the Ficus system to keep track of the number of conflicts generated and how many conflicts were resolved automatically. This paper presents the statistics gathered, which support the contention that, for important patterns of usage, the frequency of file conflicts is low enough that optimistic replication is highly attractive. Further, the statistics demonstrate that the use of automatic resolvers is both practical and important to reduce the number of conflicts reported to users.

The next section presents an overview of the Ficus file system. We begin there with an example of how a conflict can arise in an optimistic replication system, discuss the different kinds of conflicts that can arise, briefly describe how conflicts are detected, and cover other relevant aspects of Ficus. Section 3 describes the Ficus resolution architecture. It discusses the automated directory conflict resolution mechanisms in Ficus and describes how Ficus handles other types of conflicts. Section 4 discusses Ficus conflict resolver programs. It covers their interface and the various approaches used to resolve conflicts for different types of files. Section 5 presents conflict data gathered from Ficus; Section 6 discusses some related research. We close with a discussion of future work and some conclusions.

2 Ficus Overview

Ficus is a distributed file system utilizing optimistic replication [16, 4]. The default synchronization policy provides *single copy availability*; so long as any copy of a data item is accessible, it may be updated. Once a single replica has been updated, the system makes a best effort to notify all accessible replicas that a new version of the file exists via update propagation. Those replicas then pull over the new data. Ficus guarantees *no lost update* semantics despite this optimistic concurrency control. Conflicting updates are guaranteed to be detected, allowing recovery after the fact.

Ficus groups subtrees of files into *volumes*. A volume can be replicated multiple times. A background process known as *reconciliation* runs on behalf of each volume replica after each reboot and periodically during normal operation. It compares all files and directories of the local volume replica with a remote replica of the volume, pulling over missed updates and detecting concurrent update conflicts.

Several types of conflicts are possible. They include:

- Update/update conflicts
- Name conflicts
- Remove/update conflicts

The remainder of this section will discuss these types of conflicts in more detail. The next section describes how we manage these conflicts.

Since single copy availability permits any replica to be updated, even a simple partitioning of a two-replica file can result in a conflict. Figure 1 illustrates this situation. File foo has two replicas in Figure 1a, with replica 1 at site A and replica 2 at site B. If sites A and B are partitioned, as Figure 1b shows, updates to both replicas are accepted. Then, when the partition is merged, as shown in figure 1c, file foo exists in two versions. This is an *update/update conflict*.

Directories provide a special case of update/update conflicts. Partitioned creation of independent files in the same directory would ordinarily result in an update/update conflict on that directory. Since directories are internal to the file system, Ficus automatically resolves this sort of concurrent update, producing the union of all directory changes. (See [6] for a description of the algorithms employed in directory management.) A problem occurs when two files are independently created with the same name; Unix requires that each directory entry be unique. We term is kind of directory update/update conflict a *name conflict*.

Figure 2 illustrates a different kind of conflict. In figure 2a, we see two replicas of file foo before a partition. In 2b, file foo is removed at site B (indicated by the shading of site B's replica), while the partitioned replica at site A is updated. When the partition merges, as shown in 2c, if no update had occurred, then the other replica should simply be removed. However, if the updated replica is removed in this situation, the update generated during the partition is lost, possibly without the knowledge of the person making the update. Ficus' "no lost update semantics" requires that the update generated at site A not be discarded as a result of the

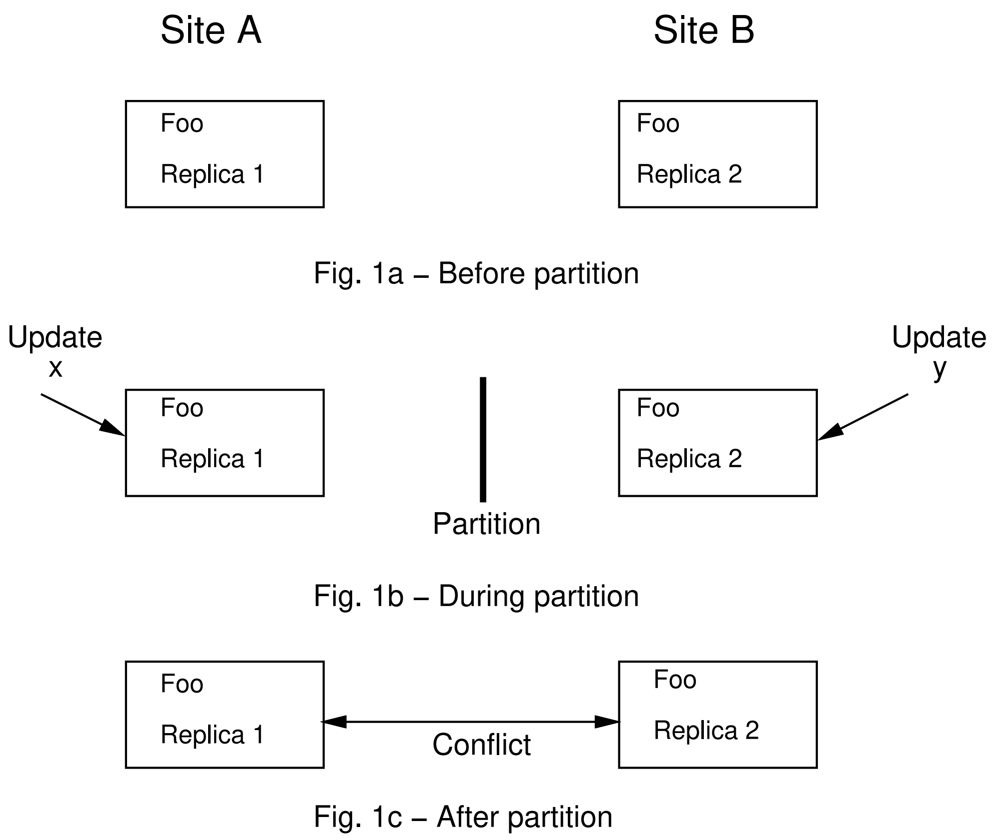

Figure 1: An update/update conflict.

removal of the file at site B. This kind of conflict is called a *remove/update conflict*.

If a file is independently deleted from two replicas of a partitioned directory, Ficus does not log a conflict. This delete/delete situation is not a problem, provided any other replicas of the file are also deleted when the partitions merge, since both deletions have precisely the same effect.

Ficus makes a "best-effort" attempt to propagate updates as they occur. However, even when no partitionings or other machine failures happen, update propagation is not guaranteed. Thus, conflicting updates can arise even without machine or network failures. Also, Ficus does not lock replicas for update even within a partition, so two replicas can accept simultaneous updates to a file that could result in a conflict. In practice, the update propagation mechanism is fast and reliable enough that conflicts unrelated to actual failures or partitioning almost never occur.

Ficus detects all types of conflicts using a mechanism known as a *version vector* [14]. Each file replica maintains its own version vector that keeps track of the history of updates to the file. Conflicts are detected by comparing version vectors from two file replicas. Version vectors reliably detect all file conflicts that involve replicas of a single file. They do not assist in ensuring the consistency of updates that span multiple files. Other mechanisms (not supported in Ficus, nor in most file systems) are required to do so.

3 Ficus Conflict Resolution Architecture

Several types of conflicts are possible in Ficus. Because of the importance of the integrity of directories, directory conflicts receive special handling. Remove/update conflicts also require some special treatment. Update/update conflicts on non-directory files are the most common case. The following subsections discuss each type of conflict and its handling in more detail.

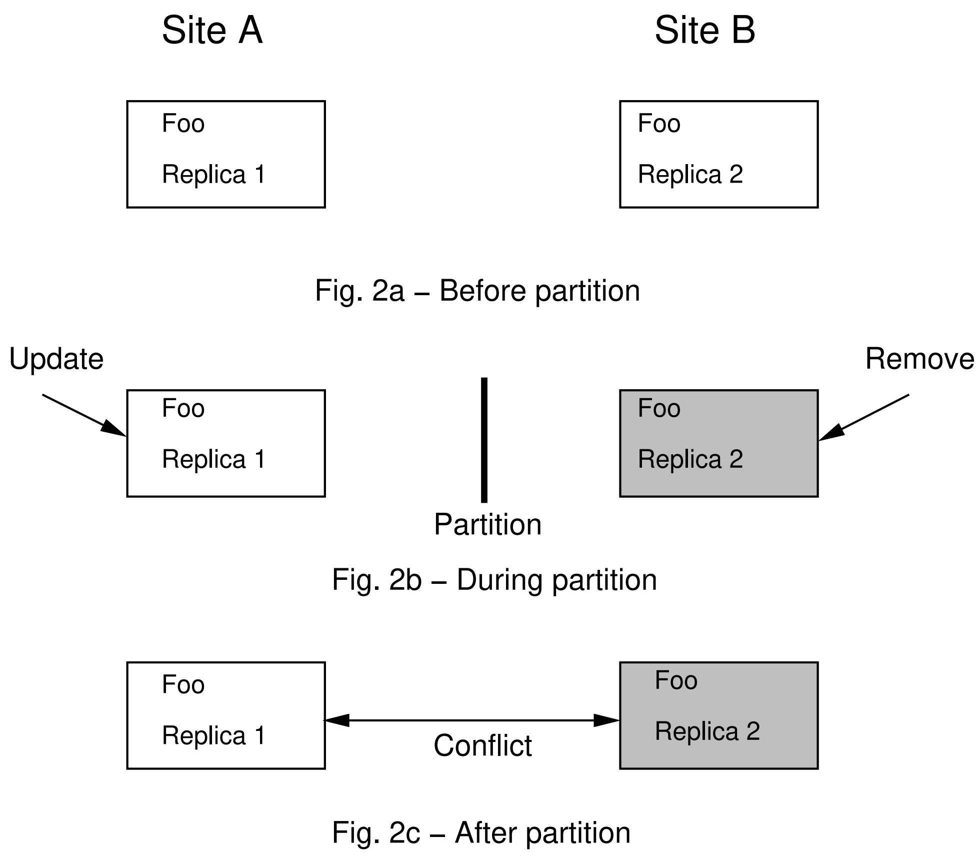

Figure 2: A remove/update conflict.

3.1 Directory Conflicts

The integrity of a Unix file system depends on its directories. If a directory cannot be used because it has received conflicting updates, a portion of the file system's name space may become inaccessible. Thus, conflicts in directories are very serious. Either they must not occur often, or they must be resolved automatically.

Ficus directory conflicts are repaired automatically during reconciliation. As shown in [4, 3], all conflicts that can occur in a Unix directory can be automatically resolved, except for name conflicts. A complete description of directory reconciliation algorithms is available in these references, so we discuss only their broad outlines here, as an example of how known semantics of files can be used to resolve conflicts.

Unix directories support only two operations: a process can add a name to the directory or remove a name from a directory. Creation of a file adds a name to a directory (in addition to creating data structures to represent the file itself). Creating a hard link to a file adds a second name for the file. File contents are discarded only when the last name for the file has been removed. While mechanically rename is an atomic operation in many Unix systems, semantically it can be treated as a remove followed by a create.

A number of issues, such as handling arbitrary patterns of failures and recoveries, distinguishing between creation and deletion of entries, and avoiding centralized algorithms, make the problem of directory management substantially more complex than it seems at first glance. In broadest principle, the automatic reconciliation mechanisms for directories examine all entries in both versions of the directory in conflict, determine which entries are common to both, and, for an entry that is present in only one directory, determine whether the file was created or deleted during partition. Ficus keeps sufficient information to distinguish precisely the patterns of file entry additions and deletions while partitioned, which in turn allows all possibly conflicting updates to be addressed.

One class of conflicting updates can create another problem, however. Concurrent creation of different files with the same name results in a directory with two identical, effectively indistinguishable names. Ficus detects that the two files are actually different, but the Unix directory model does not permit different files to have identical names, so some action is required. Ficus appends unique suffixes to each name and invokes name conflict resolvers to handle the situation. Like other conflict resolvers, if automatic resolution fails, a default resolver notifies the file owner, who must either rename or remove one of the files.

3.2 Remove/Update Conflicts

Remove/update conflicts are handled specially. Ficus is able to recognize such cases, again using version vectors. Ficus' "no lost update" semantics requires that an remove/update conflict not result in the loss of the update. On the other hand, all names for the data have been removed, so Ficus should not permit the file to remain available via those names. Ficus' solution to this problem is to move the file into a special directory called an *orphanage*. Each volume has its own orphanage directory located under its root directory. When the reconciliation process moves a file into an orphanage, electronic mail is sent to the owner notifying him, allowing him to decide whether to keep the updated file or discard it.

3.3 Update/Update Conflicts

Some update/update conflicts for non-directory files can be resolved automatically and some cannot, depending on the semantics associated with the file. Ficus has the ability to invoke various resolvers to attempt to handle file conflicts. Ficus allows individual users to specify how they would like their conflicts resolved, but also provides a default system for resolving conflicts when users have not specified their own methods, or when the user's methods fail.

3.4 Conflict Resolution

The Ficus reconciliation process runs through the files in two replicas of a volume, examining each file to determine if it has been modified since the last successful reconciliation. When it discovers conflicting versions of those file replicas, the reconciliation process marks the file as "in conflict." After marking the file, reconciliation invokes resolvers to attempt to fix the conflict. As long as the file is in conflict, normal operations on the file under its usual name will fail. Each replica of the file can be accessed by special mechanisms, and Ficus provides a tool to clear the conflict. Ficus resolver programs must use these capabilities and tools along with semantic knowledge of particular file types to resolve conflicts.

```
\.newsrc         /bin/newsrc_resolver
fortunes\.dat    /bin/fortune_resolver
man/cat[1-9]     /bin/man_resolver
\.history        /bin/arbitrary_resolver
\.bash_history   /bin/arbitrary_resolver
\.pl$            /bin/prolog_resolver
.*               /bin/default_resolver
```

Figure 3: A resolver selection file. The left column is the pattern to match against the conflicted file's pathname. The right column is the resolver that is invoked in an attempt to fix the conflict.

Ficus selects a resolver to use for a particular conflicted file by searching a personal and a system-wide resolver list. Entries in the system-wide list include resolvers for common file types. Personal resolver lists specify resolvers for file types unique to an individual. Personal resolver lists also allow an individual to choose between safety and convenience by optionally enabling resolvers that don't preserve all data. For example, some users don't care about conflicts on backup files left from editors and so have a resolver arbitrarily select one of conflicting backup files. More conservative users may wish to have data in backup files completely preserved and so may invoke a resolver saving each replica of the backup file as separate files.

Conflict resolvers almost always require knowledge about the type of the file being resolved. Since Unix systems do not provide a typed file system, Ficus infers file types from file names and from type-recognition programs that examine file contents and attributes.

The reconciliation process that examines the resolver lists matches only on file-name-to-regular-expression comparisons. Because regular expressions are used, matches can be exact or on substrings. Figure 3 shows a portion of a resolver file. Whenever a conflicted file named `.newsrc` or a file whose pathname contains `man/cat` is encountered the newsrc or manual-page resolvers are invoked. All resolvers shown in this figure have been implemented with the exception of the `prolog_resolver`.

Unfortunately, simple file name comparison cannot reliably identify all file types. For example, the file `csh.1` might be a manual page in certain contexts and shell script in others. To support more sophisticated file type identification, a resolver list might use more intelligent programs to check the file type. Continuing with the example in Figure 3, the `prolog_resolver` could abort if invoked to resolve a Perl program (whose files end with the same extension), by recognizing that certain constructs in the file were probably not legal Prolog. If a resolver aborts other resolvers are invoked in turn until one succeeds.

Separate resolver lists are provided for name conflicts and file conflicts. In retrospect, the data-specific actions taken in the case of a name conflict and a file conflict are often quite similar; only details about resolving the conflict differ. However, in certain cases it is important for a resolver to know whether it is dealing with a name conflict or an update/update conflict. In the future we plan to merge the different resolver lists and specify the conflict type as an argument to the resolver.

Ficus allows files to be replicated any number of times, so it is possible that a given file might have three or more conflicting replicas. The Ficus reconciliation mechanism works on only two replicas at a time, though, so the conflicting replicas will be dealt with in a pairwise fashion. This simplifies the writing of resolvers, since they need only deal with the common case of exactly two conflicting replicas, rather than an arbitrary number. All conflicts involving multiple replicas can be regarded as multiple pairwise conflicts, so no power is lost. Also, often not all of the conflicting replicas are simultaneously available. Since reconciliation runs between two sites known to be in communication, at least the pair of replicas they store are guaranteed to be available.

How resolvers fix conflicts depends on the semantics of the file in question. Section 4 discusses a variety of the existing Ficus conflict resolvers. Typically, resolvers read the data contained in both versions of the file, update one version of the file on the basis of both versions, then update the version vector of that replica to dominate the other, clearing the conflict.

4 Conflict Resolution Strategies and Examples

Experience with the Ficus conflict resolution mechanism has shown that there are broad classes of file conflicts that can be automatically resolved. This section discusses them, presenting examples of each. We make no claim that the list is exhaustive—in fact, we are sure it is not, since it simply demonstrates the classes of conflicts that have occurred frequently enough in our environment to draw the attention of conflict resolver writers. Further investigation, especially work in different computing environments, will undoubtedly reveal other classes of file conflicts amenable to automatic resolution.

In several important cases, much of the potential work of resolving conflicts is done by Ficus itself. As mentioned, Ficus resolves most directory conflicts automatically. Thus, any application that makes substantial use of the Unix directory structure has much of its conflict resolution problem automatically solved. Two important examples are Ficus graft points and MH mail directories.

Ficus divides its file space into volumes, each of which is connected to a single place in the file hierarchy. That place is called the volume's *graft point*, similar to a Unix mount point. The graft point must keep information about all the replicas of the volume, including each replica's storage site and other bookkeeping information. Since Ficus uses a Unix directory with one directory entry per graft point entry, graft point conflicts can be resolved automatically. For instance, if a new replica is added on each side of a partition, when the partition merges the graft point will automatically be resolved to indicate that both new replicas are available. Graft points do not experience name conflicts because the tools that update them never generate identical graft point entries.

The MH mail application also makes substantial use of directories. In MH, messages are organized into *folders*, which are implemented as directories. Most of the conflicts that could occur to MH folders during a partition are thus resolved automatically. For example, if a user re-files mail messages into different folders on both sides of a partition, the Ficus directory conflict resolution mechanism would handle most of the resulting conflicts. Only name conflicts occasionally caused by re-filing messages into the same position in two replicas of a given folder require user attention. If numeric identity of messages is not considered important, even these conflicts can be automatically resolved.

Another type of conflict that Ficus resolves automatically is conflicts on files whose contents can be automatically reconstructed. Control files used by the MH mail system are an example. These files maintain sequence and context mechanisms. They can, and often are, built as needed by MH. The only requirement is that MH generally expects something to be there—it is not prepared to deal with a totally absent file, though it can deal with a file that does not contain very useful information. Thus, to resolve conflicts on these files, the file contents are truncated. The next time MH is run, the file will be reconstructed with a default context.

Many types of files are not important to most users. For example, many users do not care about core files produced by Unix processes that fail, or about backup files produced by some Unix programs. Users who do not care about such files can put lines in their personal resolver files that either delete all such files when they get in conflict, or choose one of the conflicting replicas arbitrarily, or choose the replica with the later date. However, since some users do not want to lose some of their core or backup files without their knowledge, the system resolver file does not impose these decisions on users.

Some files are monotonically increasing logs of information. An example is the .newsrc file listing what newsgroups and articles have been read. The message numbers listed as read in each newsgroup usually increase monotonically. In the case of truly monotonically increasing logs, resolving conflicts is simple. The post-resolution version of the conflicting file simply contains the high water mark for each entry. If the file keeps exhaustive lists of items, the resolved version merges items from both conflicting versions.

In the particular case of .newsrc files, the situation is a little more subtle. The semantics of what can be changed in a .newsrc file is a bit richer than simply updating a record of articles seen. The user can subscribe or unsubscribe to newsgroups, for example. Some of these actions remove information from the file, making perfect conflict resolution impossible. The Ficus conflict resolver for .newsrc files thus must make some choices. It generally errs on the side of information preservation, presenting users with more news rather than with less. For instance, if one version of a .newsrc file indicates that a newsgroup is not subscribed to, and the other version indicates that it is, the conflict resolver subscribes the user to that newsgroup. The user can easily unsubscribe again, if that was truly what he wanted to do. If the system had left him unsubscribed when he had just recently subscribed, however, the user might not notice that his subscription had been invisibly revoked. This is an example of the create/delete ambiguity described in [5]. In some cases, taking one possible action and reporting the action taken to the user may be sufficient.

The .newsrc resolver shows a typical characteristic of many resolvers. Often, it is relatively easy to produce a resolver that is right the great majority of the time, but occasionally makes a mistake. Producing one that is right all of the time, on the other hand, may be very difficult, or even impossible. A reasonable strategy in such cases is to write a resolver anyway, as long as the resolver can do something in the tricky cases that will not produce disastrous results. If the results are merely inconvenient in the rare cases when they're not necessarily right, then the resolver has solved the conflict correctly most of the time, and caused little more trouble than not solving it at all the rest of the time. Since the alternative to this choice is notifying the user to solve it himself, this approach is attractive.

In some cases, the semantics of a file are quite simple. Score files for some of the popular Unix games are one such case. These files typically keep the top scores in sorted order. Ficus has conflict resolvers for many such games that sort and merge the two conflicting versions, removing duplicates. The case of game score files does bring up a difficulty with writing general resolvers, however. While each of the game score files contains substantially the same kind of information, the actual format is sufficiently different that writing a single resolver to handle all of them is difficult. Instead, Ficus has a class of very similar resolvers to deal with the peculiarities of each.

In other cases, conflicts can be solved simply by merging the two versions of a file into one, preserving all data in both. Doing so may cause some data to be duplicated, but many programs are able to handle such duplications without problems. One such case is the xcal program, an interactive window-based calendar manager. Conflicted xcal data files can be resolved simply by concatenating the two versions into one. The Ficus resolver includes a comment line indicating what happened, should the user care to clean up further, but the xcal program can go ahead and work with the merged version.

When the Ficus resolver files cannot resolve a conflict themselves, they call a final resolver (called the *generic resolver*) that notifies the user via electronic mail that a conflict occurred. The conflict is left unresolved until the user gives it personal attention.

In the UCLA environment, every replica of a volume is reconciled with another replica every hour. In the case of unresolvable conflicts, users might be bombarded with hourly messages about unresolved conflicts that hadn't been fixed. If a user did not log in over a weekend, fifty or more messages could accumulate in his mailbox telling him about a single conflict.

This problem is prevented by keeping track of unresolved conflicts that have been brought to the owner's attention already. When the generic resolver sends out a message about a conflict, it also logs the conflict in a per-volume conflict log file. The next time the resolver notices this conflict, it also reads the entry in the conflict log and determines that it need not send out another message to the user. The conflict log successfully limits the number of conflict report messages users receive.

However, since the conflict log is replicated in its volume (for very good reasons), this log itself can experience conflicts. Therefore, the conflict log itself needs a conflict resolver. This resolver is another example of how one can easily write a resolution mechanism that is correct almost all of the time, even though it is hard to write one that is always absolutely correct. The conflict log resolver must make sure that a given conflict is reported only once, but also that all conflicts are reported. It does so by reading both versions and writing a new version that contains all lines in either conflicted version. In case of any problems that cannot be automatically solved, the conflict log resolver simply removes both conflicting versions. Should that happen, the next time a conflict is detected a new conflict log will be created. The user will receive another message for each unresolved conflict in the volume, but no conflicts go unreported and only one extra message per conflict is sent.

Most of the existing Ficus conflict resolvers are written in Perl. Some are written in C, and some in other scripting languages. Generally, resolvers can be written in any language, provided they accept the parameters that the reconciliation process passes to them, and they return a value indicating success or failure. So far, the processing a typical resolver must perform has proven particularly suitable to Perl, in that resolvers frequently perform pattern matching, sorting, and merging, all functions that are provided by Perl. In most cases, the files in question are small, so the greater processing speed C could provide is not important.

There are undoubtedly many other types of files whose conflicts can be resolved automatically. Our approach is to first write resolvers for several known problems areas, then to write resolvers for conflicts that actually come up in practice. Thus, the set of resolvers used at UCLA gradually grows as new types of files generate conflicts and people tire of solving the conflicts by hand. At the moment, we have about 15 different resolvers, some of which are used to resolve multiple file types.

5 Data on Conflict Occurrence and Resolution

The Ficus file system has been running as the primary development environment for Ficus itself for several years. Recently, we began to gather data about the occurrence of conflicts in Ficus. This data was gathered by logging every conflict detected by the reconciliation processes and tracking conflict resolution. In addition to recording conflict detection and resolution, we also recorded the total number of updates made to determine the relative frequency of conflicting updates in our environment.

One shortcoming of this data is that most independent directory updates are *not* detected by this instrumentation. We detect all name conflicts, but do not detect the much more common case of independent creation of two differently named files. Such update "conflicts" are automatically resolved by Ficus directory resolution algorithms. We know that many such cases have arisen and have required this automatic resolution code. For example, many programs create temporary files in a user's home directory. Such programs would have created many conflicting directory updates between home-use and office machines were it not for automatic directory reconciliation. Unfortunately, this sort of conflict is not represented in our statistics, and we currently cannot precisely estimate the frequency of this situation.

The nature of the environment has a strong influence on how often conflicting updates will occur. An environment in which almost all the machines are connected almost all the time will generate relatively few conflicts. An environment in which some machines are often disconnected will generate more conflicts.

The UCLA environment contains approximately a dozen Sun workstations, each with a regular user, sharing a replicated namespace over an Ethernet. The network connection rarely fails. However, since Ficus is an experimental file system built into the kernel and undergoing continual change, the machines running it crash or are voluntarily rebooted much more often than most workstations. Machines going up and down effectively create partitions as easily as network failures do.

Two of our workstations are located at project members' homes and are only rarely connected to the network. These primarily disconnected machines store replicas of volumes important to their users. These machines and their volumes communicate with the core Ficus hosts only rarely and only to exchange updates via reconciliation. Although one might expect this pattern of usage to result in very high conflict rates, surprisingly it does not. One reason is that replica reconciliation is scheduled to coordinate the movement of the users with the data of the system, effectively allowing the system's user to act as a human "write token" [7]. While this behavior avoids many conflicts, nevertheless the conflict rate in mostly disconnected volumes is much higher than in other volumes.

Table 1 shows the conflict statistics for more than nine months of operation in the UCLA environment. About 0.0035% of all non-directory updates resulted in conflicts.

During the period under measurement, several conflict resolvers were added to our suite. Using the resolvers available at the end of the measurement period, 162 of the update/update conflicts (roughly, one in three) experienced could have been resolved automatically if the same patterns of conflicts occured today as did during this nine month period. This set of resolvable conflicts includes files related to the MH mail system, shell history and editor backup files, `.newsrc`

14,142,241 total non-directory updates
- 14,141,752 non-conflicting updates
- 489 update/update conflicts
  - 162 automatically resolved
  - 176 resolvable automatically
  - 151 not clearly resolvable automatically

98 update/remove conflicts
- 98 passed to the user for resolution

708,780 name creations
- 708,652 non-conflicting name creations
- 128 name conflicts
  - 128 passed to the user for resolution

Table 1: Conflict statistics for a nine month period. Theoretically resolvable conflicts are conflicts on files with semantics amenable to automatic resolution but for which we have not yet written resolvers.

files, and several types of game score files.

Many of the other conflicts experienced could have been handled by resolvers that have not been written. We found 176 of those, more than another third of the total number of conflicts. These include control files for the trn news reader, saved news postings, manual pages, compiler-produced object files, measurement statistics files, and score files for other games.

The remaining 151 conflicts would not be easy to resolve automatically with our current system. Files that occasionally got into conflict that cannot be resolved include such things as source code and arbitrary text files. A significant number of these conflicts occured in files placed in orphanages. Such files no longer possess their original names. Since most of our existing resolvers are selected solely by name, our current system has little hope of finding a proper resolver for these files. Explicit storage (or identification) of file type would make resolution of these files possible.

None of the name conflicts were resolved automatically. Because many fewer name conflicts occurred than file conflicts, we did not develop any name conflict resolvers in the sample period. About 0.018% of all name creations led to name conflicts, all of which were resolved by human users. On the average, each user had to deal with about ten name conflicts during this nine month period.

Taken as a whole, the average user in this environment thus had to resolve about five conflicts a month, and examine one update/delete conflict per month. In actuality, this average is misleading, since conflicts tended to happen more often to users who worked with the disconnected machines. A few users thus experienced much higher conflict rates, while many users encountered considerably fewer conflicts than the average.

The conflicts were not evenly spread across all volumes in our environment. Table 2 shows the number of updates and conflicts for different types of volumes, and the conflicts rates by volume for the nine month period.

Volumes are classified as either *shared* or *private*, and either *office*, *disconnected*, or *network*. Shared volumes indicate volumes that receive heavy update traffic from multiple users, often to different volume replicas. A prominent example of this category of volume is the games volume. Updates to the games volume involve access to shared database files (game score files). Multiple users accessing different replicas and using the same application concurrently create high probability of conflicts. In addition, the games volume is a disconnected one, meaning that replicas exist both in the office and at users' homes. Disconnection increases the likelihood of concurrent use, for the time period in which independent updates are deemed concurrent is increased. Fortunately, the shared database files have relatively simple semantics, so it is easy to write automatic resolvers for these files. Other disconnected, shared volumes included volumes of installed programs and libraries, which easily get in conflict if users are not careful about how they perform installations.

The source code volumes are another example of shared volumes, though these are stored entirely in the office. Although most source code files themselves are protected against multiple writers by a revision control service, conflicts can occur in two ways. First, multiple users can attempt to gain write permission on the same file via different replicas. Second, the same user can perform updates to two different replicas. The latter is not nearly as uncommon as it would seem, since source code volumes are replicated on server-style machines, which experience more down-time than normal workstations due to increased load. Server crashes cause automatic replica switching, creating the potential for conflicts: a user updates one replica, then switches replicas and updates the second.

Private volumes are user's personal volumes. They are almost always updated solely by the one user, and experience very few conflicts. However, when the private volumes are also disconnected, conflict rates rapidly increase. Examples of such volumes are the personal volumes of two Ficus project members who have replicas at the office and at home. Home for one user is across town, and home for the other is across the

Volume Classification	Number of Volumes in class	Number of Updates	Number of Conflicts	Conflict Rate	User-Visible Conflict Rate
Disconnected, Shared	9	1,114,855	273	.0245%	.0052%
Disconnected, Private	16	387,523	106	.0274%	.0012%
Office, Shared	27	6,316,331	66	.0010%	.0005%
Office, Private	48	6,286,754	44	.0007%	.0003%
Network, Shared	8	36,778	0	0%	0%

Table 2: Update/update conflicts grouped by volume classification.

ocean in Guam. They both call the office by modem and reconcile periodically, ranging from once a day to once every few weeks.

Most of the private volumes are rarely disconnected, and therefore one would expect there to be almost no conflicts, since only one person is performing updates and usually to the same replica. Private volumes stored only in the office accounted for only 9% of the total conflicts.

Network volumes are volumes shared between sites connected by the Internet. These volumes are all shared, in our current environment. Many of them are test volumes, leading to a low number of updates for such volumes.

As expected, disconnected volumes had a much higher rate of conflicts, 20 to 40 times as high as their office counterparts. Somewhat surprisingly, however, disconnected shared volumes suffer a lower conflict rate than private volumes.

Table 2 also indicates which of these conflicts could have been resolved automatically. For example, all of the conflicts on the game volume were on simple database score files, and therefore easily resolved. The only conflicts that the user need see in the disconnected, shared class of volumes were those in the installation volumes. Most of the conflicts on source code files in the office shared class could not be resolved automatically, however, because source code files have arbitrary semantics, and therefore require user intervention. Those conflicts that were resolvable were on object files (.o files).

The unresolvable conflict rates for disconnected volumes are still significantly higher than the unresolvable conflict rates for office volumes, but the relative difference is somewhat lower, particularly in the case of private volumes. Many of the files in disconnected private volumes that get into conflict are simple database files that can be resolved automatically. Shell histories are again a good example. In the disconnected environment, they are likely to frequently enter conflict, while they rarely enter conflict in the office environment. But these conflicts are always automatically resolvable. Once automatic resolution is taken into account, instead of one disconnected private volume update in five thousand requiring user attention, one in one hundred thousand requires it. This twenty-fold reduction in user intervention in conflicts on this class of volume is a powerful motivation for providing automatic resolution in the disconnected environment.

6 Related Work

The Ficus file system draws from several earlier systems, and has some similarities to work done by others. This section discusses some of the related work, with particular attention to that concerning optimistic replication, conflicts in optimistically replicated systems, and automatic resolution of such conflicts.

Parker's work on version vectors was an important early step in optimistic file replication [14]. It permitted reliable detection of independent updates to different replicas of a data item with limited and reasonable costs for maintaining the necessary information.

Version vectors were used in the university Locus operating system [15, 17], a system that provided data replication and dealt with partitioned operation. However, the Locus system never dealt substantively with the problems of conflicting updates.

Sergio Faissol's Ph.D. dissertation examined this question in the context of databases [2]. He investigated several classes of information that could be stored in a database, how independent updates to those classes of information could be reconciled, and the information required to perform the necessary reconciliation. His work was primarily theoretical, and was never applied to file systems.

The Coda project at Carnegie-Mellon University, discussed in detail in [18], is also developing an optimistically replicated file system. The Coda developers have considered the questions of disconnected operations in a somewhat different context than the Ficus system. They support a highly connected backbone of server machines that replicate files. While these servers may occasionally fail or become disconnected from each other, they are expected to be more reliable than the typical single-user workstation machine. Client machines cache replicas of files they actually use, and send the updates back to a server replica [10].

The nature of the Coda system makes partitioned first class replicas a less common event than in Ficus. Partitioned update is far more common between first and second class Coda replicas where simpler reconciliation algorithms are possible [10]. References [11, 12] discuss Coda's log-based approach to conflict resolution between first-class replicas. The design of conflict resolution in Coda is described in [13]. Like the Ficus approach, conflict resolvers are provided and are selected by file type.

Unlike Ficus, the Coda approach uses files that hold resolution rules that apply to all files in a directory or its subdirectories. These rules are similar in form to rules in a Unix makefile. By placing a set of generic rules in the topmost directory, Coda can achieve the same effect as Ficus' system resolver file. By using regular expressions that match only certain directory prefixes, the Ficus resolver files can achieve the same effect as Coda's per-directory rules files. Unlike the Ficus approach, Coda does not automatically serially apply different resolvers to a file in conflict, though presumably the makefile rules could be set up in such a way that they could. Generally speaking, the expressive power of the Coda and Ficus approaches seem similar. More experience with both systems is needed to determine if either approach has a clear advantage in user friendliness. The statistics presented in this paper provide the first step at addressing some of these issues.

Huston and Honeyman describe their approach to optimistic replication in disconnected AFS in [9]. This system permits updates to cached copies of data at disconnected client sites under AFS. Writes generated by a disconnected client site are logged and replayed when the client is reconnected to a server. If any of the logged write operations conflict with writes performed by some other client during the disconnection, the conflicts are detected and reported. No attempt is made to automatically resolve them, though Huston and Honeyman do briefly discuss plans to provide tools to help users resolve common types of conflicts.

Howard has developed an optimistic reconciliation-based system to permit occasionally connected machines to share files [8]. He reliably detects conflicts using a journalling mechanism, but currently makes no attempt to reconcile them.

7 Observations and Conclusions

Optimistic file replication in an environment that has any serious degree of disconnection benefits from automatic conflict resolution; it can substantially reduce the conflict rate observed by users. We present data for two environments, a usually-connected office environment and a periodically connect, usually-disconnected home use environment.

In the office environment, without automatic conflict resolution, the typical user would need to resolve around two conflicts per month, considering both update/update and name conflicts. With automatic resolution, the frequency of conflicts requiring user attention would drop to one and a half or less. The resolvers Ficus currently has installed and that will be added to our suite can reduce the total number of user-visible conflicts by about one half.

The effects are more dramatic in the home use environment. In this environment, two users generated 380 conflicts in 9 months, averaging nearly a conflict a day for each user. In actuality, one of the two users experienced the bulk of the conflicts. He made extensive use of disconnected home computing, reconciling his volumes only once a day or so, so his conflict rate was significantly higher. He observed 30 to 40 conflicts per month. Applying automatic resolvers to the home use environment reduces the observed conflict rate for this user to around seven conflicts per month.

The in-office statistics might suggest that the added value of automatic resolution of some conflicts is not that great, in that environment. However, there are some additional points to consider. First, as pointed out in Section 5, we did not gather statistics for the value of the most important case of all, the automatic, built-in directory resolver. (We hope to gather these statistics in the future.) Second, many of the conflicts that are automatically resolved are easily handled using a program, but hard to resolve by hand. If they were not automatically resolved, they would require a user to invoke a tool that might equally well be invoked automatically. Directories and binary data are examples. Third, further effort applied to writing resolvers certainly would decrease the observed rate of conflicts even more.

The case for automatic conflict resolution in less connected environments is even stronger. Environments in which disconnection is more even common than our home use environment, such as mobile computing, can be expected to have higher conflict rates. Our data suggests that conflicts in this environment are often easier to reconcile than those in the office environment. Decreasing the observed conflict rate by sixfold for a replicated home use environment is a major improvement.

Many files have semantics allowing fairly simple resolution of all conflicts. Even when not all possible conflicts a file can experience are automatically resolvable, there are often large classes of conflicts that can be fixed without human attention. Unix-style directories are one such example, where all conflicts except name conflicts can be automatically resolved. In several other cases, we have discovered that solutions that solve 80% or so of all possible problems work very well. The user need only be informed in the case of the 20% that cannot be resolved. In some cases, the resolution of the difficult set of conflicts can even be guessed at, with the user only becoming aware of the difficulty if the guess is wrong. `.newsrc` and conflict log files are two such cases.

Implementing data storage as directories offers an opportunity to leverage the Ficus directory resolution algorithms. When data follows insert/delete semantics (such as Ficus graft points do) this mapping is quite natural. In the future, we plan to restructure the directory reconciliation algorithms as a library that can be used in more general situations.

Reconciliation chooses resolvers first by file name, applying consecutive resolvers until one succeeds. Storing the file type as an attribute of the file would be a more attractive approach. Existing Unix file attributes leave little room for such information, but an object-oriented file system with a general purpose attribute service could store a resolver list as an attribute. The reconciliation process could then directly call the proper resolvers for each conflict. Such an object-oriented file system is under development in our project, and will be tested with resolver attributes. Until all data is stored as typed objects, the approach discussed in this paper offers an attractive interim solution.

While this work has been applied to a Unix-style file system, most of it is not specific to Unix systems. The general approach is applicable to many other systems and could be simplified on systems that don't allow multiple names for the same file. The approach of pairwise resolution of single conflicting files, name-based choice of resolvers, and iteratively invoking conflict resolvers until one of them succeeds appears to be generally applicable.

In conclusion, our experience with conflicts in optimistically replicated file systems is that, for one common environment, conflicts are rare. At least two thirds of those conflicts that do occur can be resolved automatically, with no user intervention or even notification. Further effort in building more resolvers would reduce the rate of user notification of conflicts even lower. Our experience with working on the Ficus system is that the typical user is not bothered by either the possibility of conflicts or their actual occurrence.

Acknowledgments

Ficus is a large system and would not be possible without the efforts of many people. In addition to the authors of this paper, Richard Guy, Dieter Rothmeier, and Steven Stovall were involved in the issues discussed in this paper. Others who have contributed to Ficus include (chronologically) Wai Mak, Tom Page, Yuguang Wu, Jeff Weidner, John Salomone, Michial Gunter, Ashvin Goel, Geoff Kuenning, Sivaprakasam Suresh, Sugata Mukhopadhyay, and Ted Kim.

References

[1] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[2] Sergio Zarur Faissol. *Operation of Distributed Database Systems Under Network Partition*. Ph.D. dissertation, University of California, Los Angeles, 1981.

[3] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System*. Ph.D. dissertation, University of California, Los Angeles, June 1991. Also available as UCLA technical report CSD-910018.

[4] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.

[5] Richard G. Guy and Gerald J. Popek. Reconciling partially replicated name spaces. Technical Report CSD-900010, University of California, Los Angeles, April 1990.

[6] Richard G. Guy, Gerald J. Popek, and Thomas W. Page, Jr. Consistency algorithms for optimistic replication. In *Proceedings of the First International Conference on Network Protocols*. IEEE, October 1993.

[7] John S. Heidemann, Thomas W. Page, Jr., Richard G. Guy, and Gerald J. Popek. Primarily disconnected operation: Experiences with Ficus. In *Proceedings of the Second Workshop on Management of Replicated Data*. IEEE, November 1992.

[8] John H. Howard. Using reconciliation to share files between occasionally connected computers. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 56–60, Napa, California, October 1993. IEEE.

[9] L. B. Huston and Peter Honeyman. Disconnected operation for AFS. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pages 1–10. USENIX, 1993.

[10] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[11] Puneet Kumar. Coping with conflicts in an optimistically replicated file system. In *Proceedings of the Workshop on Management of Replicated Data*, pages 60–64. IEEE, November 1990.

[12] Puneet Kumar and Mahadev Satyanarayanan. Log-based directory resolution in the Coda file system. Technical Report CMU-CS-91-164, Carnegie-Mellon University School of Computer Science, 1991.

[13] Puneet Kumar and Mahadev Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 66–70, Napa, California, October 1993. IEEE.

[14] D. Stott Parker, Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[15] Gerald Popek, Bruce Walker, Johanna Chow, David Edwards, Charles Kline, Gerald Rudisin, and Greg Thiel. LOCUS: A network transparent, high reliability distributed system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 169–177. ACM, December 1981.

[16] Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann. Replication in Ficus distributed file systems. In *Proceedings of the Workshop on Management of Replicated Data*, pages 20–25. IEEE, November 1990.

[17] Gerald J. Popek and Bruce J. Walker. *The Locus Distributed System Architecture*. The MIT Press, 1985.

[18] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

Author Information

Electronic mail to the authors should be directed to `ficus@ficus.cs.ucla.edu`.

Peter Reiher received his B.S. in Electrical Engineering from the University of Notre Dame in 1979. He received his M.S. in Computer Science from UCLA in 1984, and his Ph.D. in Computer Science in 1987. He has worked on several distributed operating systems projects. His research interests include distributed operating systems, optimistic computation, and security for distributed systems.

John Heidemann received his B.S. from the University of Nebraska-Lincoln and his M.S. from UCLA, both in computer science. He is currently pursuing his Ph.D. in stackable layered file systems at UCLA.

David Ratner has been a graduate student researcher with the Ficus project at UCLA for three years. His work on the project includes the rewriting and maintaining of the reconciliation, directory and file management code. Recently he has been working on algorithm and replication issues. He received his B.A. in Computer Science and a B.A. in Mathematics from Cornell University in 1991.

Gerald J. Popek has been a Professor of Computer Science at UCLA since 1973. He has been the principal investigator for the ARPA distributed systems contract since 1977. He is best known for his work on secure systems, the design of the Locus distributed Unix system, and most recently, the Ficus large scale replicated filing environment.

Popek's academic background includes a doctorate in computer science from Harvard University. He co-authored "The LOCUS Distributed System Architecture," published by the MIT Press in 1985, and has written more than 70 professional articles concerned with computer security, system software, and computer architectures.

He is a principal founder of Locus Computing Corporation. Privately-held Locus Computing is the largest independent developer of UNIX-based connectivity and distributed processing software technology.

Greg Skinner is a systems programmer for the Ficus Distributed Systems Research Group of the UCLA Computer Science Department. He received his MSCS from UCLA in 1992, with a concentration in network modeling and analysis. In his ten year professional career, he has worked on a number of projects in the areas of distributed systems and computer networks.