How to Build a File Synchronizer

Trevor Jim
AT&T Labs Research

Benjamin C. Pierce
University of Pennsylvania

Jérôme Vouillon
CNRS

February 22, 2002

Abstract

File synchronizers—user-level programs that reconcile disconnected modifications to replicated directory structures—are an increasingly familiar part of daily life for many computer users. Building a simple file synchronizer is straightforward; building one that works fast, that deals correctly with the details of filesystem semantics, and that operates robustly under a range of failure scenarios is much more difficult.

The Unison file synchronizer emphasizes portability, robustness, clean semantics, and heterogeneous operation between different types of file systems (Posix and Win32). We describe the design and implementation of Unison, discuss a range of issues facing synchronizer builders, and explain and evaluate the solutions adopted in Unison.

1 Introduction

[**Some suggestions:**

- **include a figure showing a session with Unison**
- **discuss UI**
- **describe the LWT system more**
- **Discuss: Is unison safer than rsync? does it do a better job at protecting against accidental deletions?**

]

The shift from many users per computer to many computers per user has been accompanied by a significant increase in data replication. Replicating data ensures its availability during periods of disconnected operation, reduces latency during connected operation, and protects against loss due to system failures or user errors. With this increase in replication comes, inevitably, the issue of how to combine updates to different replicas. In tightly coupled systems, this issue can be addressed by conservative schemes that preserve "single-replica semantics." However, in larger systems and systems whose parts must often operate in disconnected mode, optimistic strategies must often be employed to achieve acceptable performance and/or availability. These schemes allow concurrent updates to the replicated data, which are later reconciled by propagating non-conflicting changes to all replicas and detecting and resolving conflicting changes.

The tools and system components that perform this sort of reconciliation, often generically called *synchronization* technologies, come in many forms—distributed operating systems, distributed databases, application middleware layers, PDA hotsync managers, laptop file synchronizers, etc. Our primary interest here is the subcategory of *file synchronizers*—user-level tools whose job is to maintain consistency of replicated directory structures.

Engineering a file synchronizer is a challenging task, for several reasons. First, a file synchronizer must deal with all the low-level quirks of filesystems. Second, file synchronization is an inherently distributed task, requiring careful attention to efficient network utilization and demanding robust operation in the face of a range of possible host and network failures. Finally, the design of a user interface for a synchronizer takes careful thought, since it must present a potentially large amount of information about updates, conflicts, failures, etc., in a clear and intuitive way.

We have designed and implemented a file synchronizer called Unison.[1] From modest beginnings in 1995, Unison has grown into a mature tool with a significant user community[2] and high ambitions in the areas of portability, robustness in the face of failures, and smooth operation across different filesystem architectures.

In the process of building Unison, we have learned many lessons about the challenges of file synchronization and experimented with numerous ways of addressing them. The goal of this paper is to record these experiences, and the ultimate design that arose from them.

We begin in Section 2 by discussing the major choices that have guided Unison's design. Section 3 describes the structure of the implementation. Section 4 explores in more depth several critical implementation issues related to robustness. Section 5 describes further refinements related to cross-platform synchronization. Section 6 discusses performance issues, and Section 7 presents some preliminary measurements of Unison's performance. Section 8 sketches a number of useful extensions of the synchronizer's core functionality. Sections 9 and 10 discuss related and future work.

2 Design

The most important goal of any file synchronizer is safety. A synchronizer changes scattered and potentially large parts of users' filesystems, which may contain sensitive and valuable information. Moreover, this work is largely unsupervised by users. This puts a synchronizer in a unique position to do harm, and places a heavy responsibility on synchronizer implementors to ensure fail-safe behavior in all situations. Doing so requires several different sorts of bulletproofing, which we describe in Section 4.

An issue closely related to safety is the treatment of conflicts. All synchronizers try to propagate non-conflicting changes between replicas, ideally making them equal at the end of the run, but designs differ in what happens when this is not possible. Some[3] insist on consistency—all replicas must be identical after synchronization, even if this means that some of the changes made by the user to one or another of the replicas must be discarded or overwritten. Others treat the user's changes as sacred: if a file has changed in incompatible ways in two replicas, then the synchronizer must do *nothing* to this file without guidance from the user, even though this means that the two replicas will differ after synchronization. Unison adopts the latter point of view.

Another major issue in the design of any synchronization technology (not just file synchronizers) is what sort of information the synchronizer can see about the changes to the replicas.

- *Trace-based* (also called *log-based*) synchronizers detect updates by examining a complete trace of all file modifications. The trace may be provided by the operating system to the synchronizer when it runs, or the synchronizer itself may be watching modifications in real time. Distributed operating systems, databases, and application middleware layers fall into this category.

- *State-based* synchronizers use only the current states of the file system to detect updates. This may involve examining modtimes, inodes, dirty bits, file contents, etc., and comparing them with saved copies or summaries. Hotsync managers and user-level file synchronization tools fall into this category.

Trace-based synchronizers have more information to work with, so in principle they can make better decisions about how to propagate changes and handle conflicts. However, they require support for synchronization to

[1] Unison's source code is available under the GNU Public License. It can be downloaded, along with precompiled binaries and documentation, from `http://www.cis.upenn.edu/~bcpierce/unison`.

[2] It is difficult to get a precise idea of the number of users of an open-source tool. Our server log records downloads from about 10,000 distinct IP addresses; this gives a high estimate. (It is probably a substantial overestimate, but even this is not certain. Precompiled versions of Unison for several operating systems besides the officially supported ones [Red Hat Linux, Solaris, and Windows] are offered by civic-minded users; Unison is also packaged with the Debian and Suse Linux distributions.) A low estimate of 500 is given by the size of the email discussion and announcement lists. The biggest single user we know of synchronizes 18Gb of programs and data nightly between geographically separated servers. It is also used by several organizations as the core of automated backup and file sharing services for multiple users, and by other organizations for distributed web site maintenance.

[3] E.g., the Windows 2000 file replication service. IceCube [SRK00, KRSD01] adopts a similar point of view.

be built into systems at a very basic level. State-based synchronizers are more portable, can be run as user-level programs without administrative privileges, and can be used in settings involving multiple organizations or administrative domains where full-blown distributed filesystems and databases are impractical.

One of our goals for Unison was that it should be portable across a range of operating systems, including recent releases of Windows and all popular Unix variants (Linux, Solaris, OS X, *BSD, etc.). This argued strongly for making Unison a user-level, hence state-based, tool—i.e., designing it to run as an ordinary, user-level application program, rather than being part of the operating system or relying on special operating-system hooks. Besides portability, there were two other reasons for this choice. The first was simplicity: as noted above, user-level synchronizers are much more straightforward to build and install than synchronization components of distributed filesystems, since they stand above the rest of the filesystem and interact with it through the same APIs as other user programs. Second, user-level synchronizers are a very common category of tools in the real world, but one that has so far received relatively little attention from the research community.

Another goal was to offer reasonable performance for synchronizing large replicas (around a gigabyte—the size of many people's home directories these days) over a variety of communication links, from fast local ethernets to 56K modems and DSL lines. One immediate corollary of this decision is that update detection (which involves scanning the whole of each replica) must be performed locally at each host, that any large auxiliary data structures saved between synchronizer runs must be replicated at each host (and must therefore themselves be kept carefully in sync), and that the information exchanged during update detection should concern only changed files.

A final important goal was that Unison should come with a clear and precise specification—one clear enough to put in the users' manual and precise enough to permit users to predict its behavior in all situations. Thus, from the beginning, experimenting with ways to formalize Unison's features has gone hand in hand with the engineering of the tool, and the intuitions gleaned from this effort have contributed enormously to the success of the pragmatic side of the project. However, we will not say much about the specification here, since it has been described in detail elsewhere; see [BP98, PV01] and, for an extension of our approach [RC01].

The success of a project can depend almost as much on its *non*-goals as on as its goals. The most important non-goal for Unison so far has been synchronizing information *within* individual files. If a file has been changed in different ways in the two replicas, Unison signals a conflict, even if the file is a database and the changes were to different records. Of course, there is a class of tools—generally called *data synchronizers*—that synchronize information between databases at the record level. For example, the "hotsync manager" that moves data between a Palm or other PDA and a workstation is a data synchronizer. But most data synchronizers are able to operate *only* on flat collections of databases, not on hierarchical file systems. (PumaTech's IntelliSync is one exception.) We believe that many of the ideas in Unison (and perhaps some parts of the implementation) can also be applied to data synchronization, and we are currently investigating this possibility.

Another feature that is found in some synchronizer designs but is omitted from Unison is the ability to invoke sophisticated heuristics to help resolve conflicts. When Unison detects a conflict, it simply asks for guidance from the user. (Recent releases of Unison take a small step by providing hooks to invoke an external merge program.)

A final non-goal of the current implementation is multi-replica synchronization: Unison only synchronizes two replicas at a time. Unison can synchronize n replicas by performing several (more precisely, $2n - 3$) pairwise synchronizations, but this mode of use does not scale gracefully beyond about $n = 4$. We would like to implement multi-replica synchronization in the future, but doing this well will require additional mechanisms (version vectors, gossip architectures, etc.) that will substantially complicate both the design and the implementation.

3 Architecture

Unison uses a simple client-server architecture, as shown in Figure 3. The connection between client and server processes can be established in two different ways. In *socket mode,* both processes are started manually, and the client connects to the server on a predetermined port. In *tunneling mode,* the client process is

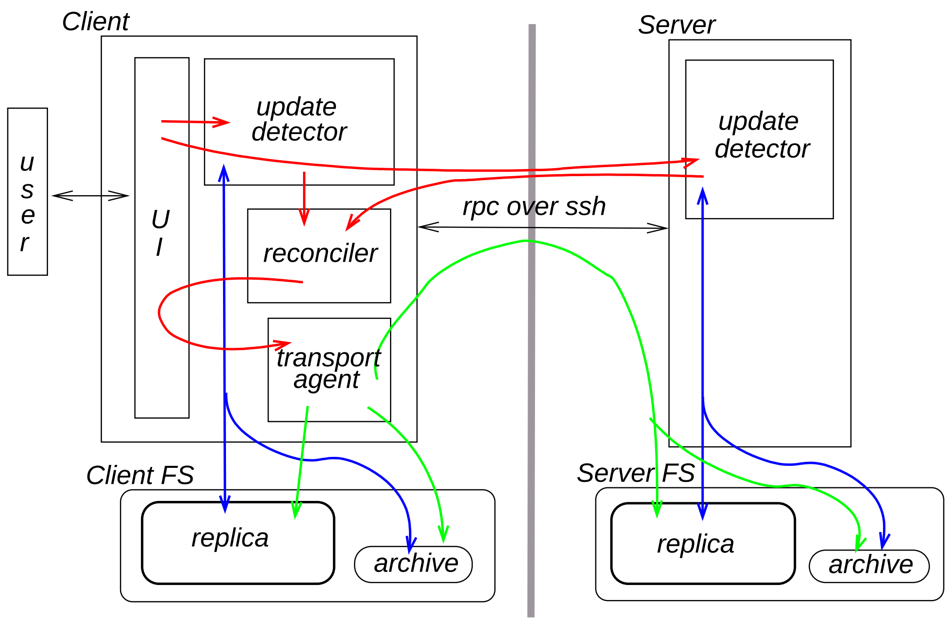

Figure 1: Unison's architecture

started first; it uses `ssh` to open a connection to the server machine, starts the server process there, and communicates with it using pipes. The second method is the recommended one, since the use of `ssh` makes the communication between client and server secure from eavesdroppers and other ill-doers. The first method is provided for users who cannot use `ssh`, e.g., because their server machine is running a version of Windows for which no `ssh` is available.

Most of Unison's functionality is concentrated on the client side: the server process is used only for detecting updates and helping propagate new versions of files. Client and server communicate by synchronous remote procedure calls.

When the client process is started, two *roots* for the synchronization are specified, either as command-line arguments or by including them in a permanent preferences file. Each root specifies a starting point for synchronization in the corresponding host's file system.

The first step of synchronization, *update detection*, is performed separately by the client and server processes. Each process reads an *archive* file that was created at the end of the last synchronization; this file contains modification times and other metadata (such as inode numbers when running under Unix), as well as a cryptographic fingerprint of the contents of each file. These are compared to the current states of the two filesystems to produce a list of the paths within each replica whose contents have changed since the previous run of Unison. (Note that we do *not* consider a file to have been updated if it has been touched but its contents have not changed. This is a significant design choice, since it means that a file that is touched in one replica and changed in the other will be treated the same as if the first replica had not been touched—i.e., the change from the second replica will be copied to the first, rather than signaling a conflict.) The server process sends its list of updates back to the client.

In the second step, called *reconciliation*, these lists are merged to produce a *task list*—a list of changed paths, each annotated to show which replica's contents should be copied to bring the two into agreement. (We consider "absent" as a special kind of contents, so that deleted files, created files, and changed files can be handled uniformly at this step.) The critical design issue here is the handling of conflicts. A path is said to be *conflicting* if (1) it has been updated in one replica, (2) it or any of its descendants has been updated in the other replica, and (3) its contents in the two replicas are not identical. In particular, we consider deleting a directory (or replacing it with a file) in one replica and changing a descendant of that directory in the other replica as conflicting operations. Paths whose contents have been changed in both replicas are marked as conflicts—i.e., they are added to the task list, but with no default recommendation for which replica's copy should be propagated to the other—*unless* the new contents happen to be the same on both sides. In this case, the files are marked as successfully synchronized and omitted from the task list.

Next, Unison displays the task list in the user interface and waits for confirmation. (Two different user interfaces are provided: a textual interface using a plain terminal and keyboard, and a Gtk-based GUI. The desired UI is selected at start-up time by a command-line switch or a permanent preference setting.) The user may now examine Unison's recommendations for changing the replicas, manually cancel or override them if desired, and instruct Unison what do to with conflicting paths. Unison may also be run in batch mode, in which case no confirmation is needed at this step and conflicting paths will simply be skipped.

Next, changes are *propagated* between the replicas as specified by the (user-modified) task list. (This step is discussed further in the following section; some care is needed to ensure that the replicas will not be left in a bad state if a failure occurs during this phase.)

Finally, Unison updates the archive files on both client and server to reflect the new contents of the paths that were successfully synchronized on this run.

4 Robustness

Users place a great deal of trust in file synchronizers. They use them to update important files with little or no supervision; some even use them as primary backup utilities. Avoiding data corruption is a top priority. In this section, we discuss the protective measures that Unison takes to avoid losing user data in the face failures. (Unfortunately, it is impossible, in practice, to protect against all failures. For example, Unison's one—to our knowledge—catastrophic failure occurred when a user's RAID setup failed and behaved as though all its files had been deleted. When the array was synchronized with another replica, the deletions were propagated! Fortunately, the data had been backed up.)

Crash resistance A run of Unison can be interrupted for many reasons: systems can crash, disks can fill, phone and power cords can be pulled out, and impatient users can abort lengthy transfers. To make sure that data is not lost when these things happen, we work to maintain the following invariant:

> At every moment during a run of Unison, every user file has either its *original* contents, or its correct *final* contents.

With this invariant, any interrupted sync is a correct *partial* sync, and can be completed simply by running Unison again.

Unfortunately, file system limitations prevent us from doing more than approximating this invariant. What we need is the ability to replace one file with another in an atomic step. In most file systems, this isn't possible, or is possible only under limited circumstances. For example, Posix specifies that a file-to-file replacement be atomic; however, we find that in practice, file systems don't achieve this if the files are on different partitions or different drives. Posix does not allow directory-to-file or file-to-directory replacements at all. Windows file systems do not support atomic replacement of any sort.

Unison uses a two-phase commit to approximate atomic file replacement when it is not provided by the file system. In the first phase, the file to be replaced is renamed to a temporary file, and in the second phase, the replacement file is renamed to the target file. The temporary file can be deleted after the commit. If Unison is interrupted in the middle of this process, the invariant will fail—the original file has not been lost, but it has a different name. The next time Unison is run, it notices that it was interrupted while simulating the atomic replacement, and it notifies the user of the situation.

Atomic file replacement is needed in two places: when Unison updates a user file, and when Unison updates its own internal state (the archive).

When Unison needs to update a user file, it first transfers the new version from the other replica to a temporary file. Since this involves network traffic, building the temporary file is not atomic. Once the file is successfully transferred, Unison uses atomic replacement (or simulates it) to move the temporary file to its final location.

Unison updates its own archive in much the same way: it first writes the new archive to a temporary file, then uses atomic replacement (or its simulation) to replace the old archive. Unison might be interrupted after it has updated a user's file, but before it has a chance to update its archive. This is no problem: the next time Unison runs, it will detect that the file has changed in both replicas, notice that the contents are now equal, and note that the files are in sync when it updates its archive at the end of the run.

Users sometimes run more than one invocation of Unison at once, by mistake. Therefore, we are careful to prevent concurrent updates to archives using a lock.

Concurrent file system updates Users do not like to wait for synchronizers—they continue to modify files while the synchronizer works. There are two cases to watch out for. First, the user might modify a file after Unison decides that it should be replaced with the version from the other replica but before it does so; Unison checks for this by re-fingerprinting the file it is about to overwrite immediately before it does so (after the new contents has been transferred to a temporary file on the same machine). Second, the user might modify a file while Unison is in the middle of transferring it to another system; Unison guards against this by checking that the temporary file on the remote system has the expected fingerprint after the transfer. If either case is detected, Unison signals a failure for that file.

Once again, file system limitations prevent us from providing complete safety. For example, a user might open a file that Unison is about to replace. After Unison replaces the file, the user can write to the old file. However, that file has been unlinked, so when the user closes the file, the changes will be lost. Unlike Unix, Windows actually prevents the deletion of a file that is open; for safety, that may be the better choice.

Update detection Update detection is a safety critical task: if a file has changed and Unison does not detect this, it might mistakenly replace the changed file with a different changed file from the other replica.

Modtimes are not sufficient to detect updates, because they do not change when a file is renamed. For example, if `foo` and `bar` have the same modtime, and `foo` is deleted and `bar` is renamed to `foo`, modtimes would indicate no change to `foo`. Unix (but not Windows) has inode numbers, which can detect this situation. But even modtimes plus inode numbers are not 100% safe, because modtimes can be set back (e.g., when backups are restored).

The only sure way to detect updates is to use fingerprints: Unison computes the fingerprint for each file and compares it to the last known fingerprint, which is stored in the archive. However, fingerprints are slow, and users don't like to wait. For impatient users who are willing to risk missed updates under certain conditions, we provide a less-safe update detector based on inode numbers and modtimes for Unix, and an even-less-safe update detector based only on modtimes for Windows.

Transfer errors Our file transfer protocol uses checksums to make sure that files are transmitted correctly between computers. This is quite important, because we use complex transfer protocols (like a threaded rsync—see Section 6) for efficiency. The checksum protects against bugs in our protocol implementation as well as network failures.

Archive loss Unison's archive files are just ordinary files that can be deleted by the user; we are careful to make sure this does not cause any synchronizer failures.

If Unison finds that its archive files have been deleted, or cannot be read, it takes a conservative approach: it behaves as though the replicas had both been completely empty at the point of the last synchronization. Files that exist only in one replica will be propagated to the other, files that exist in both replicas and are unequal will be marked as conflicting, and files that exist in both replicas and are equal will be marked as synchronized. The user will have to reconcile the conflicting files by hand.

Error handling When Unison encounters what looks like an intermittent error (e.g., a corrupted file transfer), it displays an error message on a per-file basis. If Unison encounters a catastrophic failure (e.g., a full disk), it simply quits. This is safe: the measures we have taken for crash resistance ensure that no user data is corrupted.

5 Cross-platform synchronization

Different operating systems often have different file system conventions. For example, they may have different limits on the length of file names, or different models of file ownership and access control. A cross-platform file synchronizer that does not take these differences into account can confuse its users, or even corrupt their files.

Consider, for example, case sensitivity. In Unix file systems, file names are case sensitive: a directory can hold two different files with names `foo` and `FOO`. In Windows, on the other hand, file names are case *in*sensitive: a directory cannot hold two different files named `foo` and `FOO`, and if `foo` exists, it can also be referred to as `FOO`. Windows displays the file name as it was capitalized when created, but disregards case when accessing the file. If a file synchronizer ignores this detail, it can cause a file to be lost. To see how, suppose a file `foo` has been synchronized between Unix and Windows, and subsequently, `foo` is deleted and `FOO` is created on the Unix file system. On the next synchronization, if the file synchronizer does not realize that `foo` and `FOO` refer to the same file under Windows, it may try to: (1) copy `FOO` to the Windows file system; (2) delete `foo` from the Windows file system. In Windows, (1) will overwrite the file `foo`, and (2) will delete it, so that there is no file `foo` *or* `FOO` on the Windows file system after synchronization.

Unison resolves differences in file system conventions by using a lowest common denominator approach: it only synchronizes information that makes sense on both file systems. Sometimes, this means that a file or file attribute cannot be synchronized at all; if so, Unison displays a message to this effect. In other cases, some useful information can be passed in one or both directions.

We illustrate this below by examining some differences between Unix and Windows, and showing how we resolved each of them in Unison.

Case sensitivity of file names Our solution to the case sensitivity issue described above is to treat *both* file systems as case insensitive when synchronizing between Unix and Windows hosts.

When Unison finds two Unix files in the same directory whose names differ only in their capitalization, it displays a message saying that the files cannot be synchronized to the Windows system, even if there is a file with one of the names on the Windows system. For example, if there are files `foo` and `FOO` on the Unix side, and there is a file `foo` on the Windows side, we refuse to synchronize the Unix `foo` and the Windows `foo`. This is because we consider `foo` and `FOO` to be the same name—hence, there is no reason to pick the Unix `foo` over the Unix `FOO`.

When Unison finds a file `foo` on the Unix side and a file `FOO` on the Windows side, it will synchronize their contents—we are careful to use a case insensitive string comparison to decide what files match up. This is important not just in Unix-Windows synchronizations but also in Window-Windows synchronizations. We also use case insensitive comparisons in deciding what files to ignore (section 8): in a Unix-Windows synchronization, it is not possible to ignore a Unix file `foo` while at the same time not ignore a file `FOO` in the same directory.

We do not currently synchronize the capitalization of filenames. For example, if `foo` is synchronized between Unix and Windows, and it is renamed to `FOO` on one side, that change is not propagated by Unison. In the future we plan to handle this by treating capitalization as a file attribute (like permissions—see below).

Finally, we have to be careful when copying a file from one replica to another. For example, suppose we want to replace the contents of a file `foo` on one system with the contents of file `FOO` on another. We must be sure to use the target name `foo` and not `FOO`. Otherwise, if the target is on a Unix system, we would end up with two files, the new `FOO` and the old `foo`.

File permissions In Unix, a file has an owner and a group, and it is possible to specify read, write, and execute permissions separately for the owner, the group, and all others. Windows 98 (FAT32) files do not have owners or groups, and it is only possible to specify that a file is read-only or read-write. This means that owner, group, read, and execute permissions cannot be synchronized from Unix to Windows 98; we only synchronize write permissions.

There are two separate cases to consider: creation of a file, and changes in permissions. When a newly created file is propagated to a remote replica, the permission bits that make sense in both operating systems are also propagated. The values of the other bits are set to default values (they are taken from the current umask, if the receiving host is a Unix system). If a Unix file permission changes, we only propagate it if it would change the corresponding Windows permission.

Windows NT, 2000, and XP filesystems have a more sophisticated access control system; we have not yet implemented synchronization of file permissions on these systems.

Symbolic links Windows does not have symbolic links (shortcuts do not work in the same way). When Unison encounters a symbolic link in a Unix to Windows synchronization, what it does depends on whether the user has chosen to treat the link as transparent or opaque (see section 8). If the link is considered opaque, Unison will display a warning that the link cannot be synchronized. If the link is considered transparent, Unison will synchronize whatever the link points to.

Illegal file names Windows forbids many file names that are allowed in Unix, for example, file names containing certain characters (e.g., colon, backslash), and file names that conflict with Windows device drivers (e.g., con, prt). Unison displays a warning message when such file names are found on the Unix side, leaving it up to the user to change the name of the file if desired.

Modtimes In Windows, file timestamps have a 2 second granularity, while in Unix, they have a 1 second granularity. This is not a difficulty for update detection, because clocks are only compared locally. However, it does mean that when synchronizing modtimes between systems, the low-order bit of the modtime must be ignored.

Line endings In Windows, text files use a newline and carriage return to end lines, while in Unix, lines in text files end with just a newline. Unison does not currently translate line endings when synchronizing between Windows and Unix, but this is an oft-requested feature.

The main difficulty in implementing the line ending translation is in the reconciliation phase. When line endings are being translated, a Unix text file will have a different length and contents than a corresponding "identical" Windows file. This won't affect update detection, because updates are calculated locally for both replicas. However, the reconciliation phase must compare files from both systems, taking each system's line ending conventions into account. For example, if a file is modified identically on both the Windows and Unix file system, the reconcile phase should not indicate a conflict for the file.

This issue can be addressed by using fingerprints calculated by scanning the file in text mode on each system. Using text mode to scan the file eliminates the difference in the line endings, so that "identical" files have identical fingerprints.

Cross-platform file systems Samba can mount the file system of one operating system onto another. When this happens, the file system semantics of the remote mount will be different from the rest of the file system: it may have different case sensitivity, for example.

To be completely bulletproof, Unison should check for this situation. However, there is little operating support to do this efficiently. For example, there is no system call to find out the maximum length of file names in a (possibly remotely-mounted) directory. The best solution that we can think of is to create dummy files in the directory with longer and longer names, until an error is encountered. This is neither robust (a file create can fail for many reasons other than a long file name, e.g., lack of write permission) nor efficient.

6 Performance

Network file transfers are expensive, so we have taken several steps to improve their performance.

Threads The Unison client and server communicate using a synchronous RPC protocol. We used a synchronous protocol because they are easier to use than asynchronous protocols, but they can be slow, particularly when there are many round trips. In fact, a file transfer in Unison does require a number of round trips:

	Unison	Rsync
Single machine	14 s	5 s
Local network	16 s	10 s
Internet	4 min 20	4 min 05
Internet, with ssh compression	1 min 15	1 min 05
Internet, single-threaded	17 min 30	

Figure 2: Unison vs. rsync, propagating many new medium-sized files

- The data must be transmitted.
- We must check[**awkward**] that the file was not modified on either the client or server before putting the file in place, to guard against concurrent updates.
- Once the file is successfully updated, a round trip is required so that both the client and server know this and can update their respective archives.

To avoid the penalty of the round-trip latency, we use a small number of cooperative threads to transfer several files in parallel. We chose to use cooperative threads because they are not subject to race conditions, and because they are more portable than native threads. Threading gives a substantial performance increase, as demonstrated by the measurements in the next section.

Rsync Unison includes an implementation of the rsync protocol [TM96, Tri99], which speeds the transfer of a file from one system to another when there is a similar file on the target system. This is often the case in synchronization, where the two files start out as identical and one is edited slightly.

Fingerprinting A fingerprint is a collision-resistant hash of a file; if two files have the same fingerprint, then it is very unlikely that their contents are different. We use fingerprints in update detection, but they can also speed file transfers, by making them unnecessary: before transmitting a file, we send its fingerprint to the other system, where it is compared to the file that we want to replace. If the fingerprints are equal, presumably the files are equal and we don't have to transfer the file. This can happen if both files have been changed identically, or, more commonly, when two identical replicas are synchronized for the first time.

7 Measurements

In implementing Unison we have focused mainly on robustness and following a clear specification. We have included a few features like fast update detection algorithms and an rsync-based update transfer protocol that make orders of magnitude differences in performance, but the tool's low-level performance has not been extensively tuned. Nevertheless, we present a few benchmarks here to give a rough idea how Unison performs in comparison with related tools.

The rsync utility [TM96, Tri99] is a mature, robust, and widely deployed tool, and it is used to address some of the same sorts of replication tasks that are Unison's bread and butter. It uses the same transport mechanisms (raw sockets, or tunneling over ssh) as Unison's, so performance comparisons are easier to interpret.

The first set of measurements, in Figure 2, compares the simple file transfer capabilities of Unison and rsync by using both to transfer a collection of medium-sized files (6000 files, totaling 16.6 MB) under different network configurations. We tested only the ssh-tunneling mode, since the insecurity of the raw sockets mode in both tools limits its applicability. We ran the tests from a machine in France (Pentium III 800Mhz) to (1) another replica on the same machine, (2) another machine on the same local network (Pentium III 500 Mhz), and (3) a remote machine server (Sun Enterprise 3000, UltraSPARC 4×250Mhz) in Philadelphia. The local network was a 100Mbits/s Ethernet. The latency over the Internet connection was about 130ms (round-trip), and the bandwidth was about 150KB/s.

	(Full transmission) Uncompressed	With ssh compression	Small change
One file	80 s	30 s	6 s
Two files	135 s	35 s	9 s
Three files	220 s	50 s	12 s
Four files	290 s	70 s	15 s

Figure 3: Effects of rsync algorithm on large file transfers

For the internet connection, we give three separate numbers: the first line shows the transfer times for Unison and rsync using ssh with its default parameters. The second shows the same transfers with ssh compression enabled. (By way of comparison, gzipping the files used for the test reduces their size by about a factor of five.) The final line shows Unison's performance when the multi-threaded transfer feature described in Section 6 was disabled; this measurement makes it clear that threads are critical to achieving good results over a high-latency link.

Unison is much slower than rsync on a single machine and on a local network. We conjecture this is mostly due to the fingerprints that Unison computes for robustness (cf. Section 4)—checking that a file is unchanged before propagating it, then checking after transfer that the copy is not corrupted. At about 1Mb per second, Unison's performance is still acceptable for local transfers.

For internet transfers, Unison and rsync perform comparably. There is a huge performance improvement with compression, showing that Unison is bandwith-limited, despite the high latency ($130\,ms$ round-trip) of the network connection.

The next set of measurements, in Figure 3, shows how Unison's implementation of the rsync algorithm improves the propagation of small changes to large files. This is an important case for Unison (e.g., large mailboxes with a few messages added to the end fit this pattern). The figure shows total transmission times for one, two, three, and four large (4.5Mb) files after different kinds of changes. In the first two columns, the contents of the file are completely changed, so that the rsync algorithm is not helping at all; we show both the uncompressed transmission time and the time with ssh compression enabled. For the third column, a single line was added to the end of the file—an ideal case for the rsync algorithm.

When there are more than two files, the full transmission time grows linearly. Sending only one file is proportionally slower since, in the current implementation, the contents of each file are transferred sequentially using synchronous RPC. As might be expected, the rsync algorithm gives a huge improvement for files with small changes.

The standard rsync program is much faster than Unison for this task: it takes about 1.5s to transfer three files with small changes. We have two ideas for how this difference can be explained. First, Unison's implementation of the rsync algorithm has not been carefully tuned. Second, the current implementation sends huge checksum tables, while rsync trims the checksums to provide just enough information to prevent the probability of transmission failure from becoming too high.

Typically, between runs of Unison only a few files will have changed, relative to the total size of the replicas. In this case, update detection takes a large proportion of the total synchronization time. Getting good numbers for update detection times turns out to be quite tricky: times can vary widely depending on how scattered the inodes are on the disk, how many are cached in memory, etc. However, Figure 4 gives a rough indication, showing ballpark times for update detection on replicas of three sizes. (Incidentally, Unison's archive file for the large replica was about 6Mb. Loading it took about 2s at the beginning of each run.) We expect Unison's performance on large replicas with few changes to scale better than rsync's, since the checks performed by Unison are local to each host, while rsync requires some communication for each file.

8 Refinements

This section discusses a number of useful refinements to the core functionality described above.

	Total size	Files	Update detection time
small replica	12Mb	2K	2-10s
medium replica	400Mb	50K	30–90s
large replica	2Gb	100K	2–5 minutes

Figure 4: Update detection times for various replica sizes

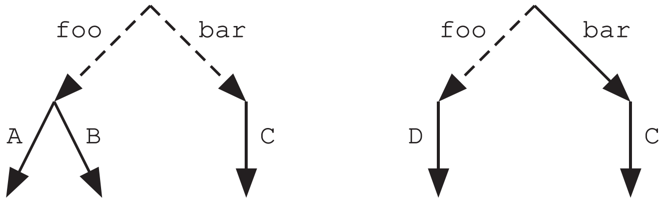

Figure 5: Two synchronized replicas. Links are displayed as dashed arrows, and non-links are displayed as solid arrows. Link foo is treated opaquely: the contents of the directory it points to are not synchronized between the replicas. Link bar in the replica on the left is treated transparently; it is synchronized with a directory (not a link) in the replica on the right. The contents of directory bar on the right are synchronized with the contents of the directory pointed to by bar on the left.

File permissions Unison can synchronize the permissions of a file as well as its contents. For security reasons, however, we don't synchronize the Unix `setuid` and `setgid` bits, or the Unix owner and group ids by default (because an id can correspond to two different users on two different systems). All files are created with the owner and group of the Unison process.

Read-only permissions require some care. For example, if a directory and its children are changed from read-write to read-only, then we synchronize the permissions of the children before we synchronize the permissions of the directory itself. If this were done in the opposite order, Unison would fail when it tried to change the permissions of the children.

More information on how we synchronize permissions between operating systems with different access control systems is given in Section 5.

Ignoring files Users often find that their replicas contain files that they do not ever want to synchronize—temporary files, very large files, old stuff, architecture-specific binaries, etc. Unison lets users specify what files it should ignore using regular expressions.

Symbolic links Ordinarily, Unison treats symbolic links in Unix replicas as "opaque": it considers the contents of the link to be just the string specifying where it points, and it will propagate changes in this string to the other replica. This works well for a link that points to another place within the replica, or for links that point to files and directories that are outside the replica, but are replicated on the two systems (by Unison or by some other means).

Sometimes, it is useful to treat a symbolic link "transparently," acting as though whatever it points to were physically *in* the replica at the point where the symbolic link appears. For example, if a symbolic link is synchronized opaquely with a Windows replica, then the file(s) accessible through the link on the Unix replica won't be available on the Windows system. Instead, we would like the Unix link to be synchronized with a Windows copy of the directory that the link points to. Opaque and transparent links are illustrated in Figure 5.

Unison lets users specify whether a given link should be treated opaquely or transparently (using regular expressions).

Modtimes Unison can synchronize modification times of files, but this is not always a good idea. For one thing, clock skew is inevitable between any two computers. A file changed in the slow replica can cause its modtime in the fast replica to move backwards. Modtimes that move backwards can confuse software that uses them to detect when files change, like make, or Unison itself. (Unison can cope with this by using fingerprinting to detect updates rather than modtimes, but this is expensive.)

Synchronizing modtimes for directories is also quite difficult. In Unix, a directory's modtime changes every time a file in the directory is touched. So, when Unison synchronizes a file in a directory, the modtime of the directory is disturbed; if we want to synchronize the modtime of the directory, we must re-set it. This in turn disturbs the modtime of the parent of the directory; and so on. It is hard to get this right, and it is hard to do it efficiently; for those reasons, we never synchronize directory modtimes in Unison.

A final problem is that different operating systems may have different clock granularities; Section 5 explains how we handle this.

9 Related work

A sizable body of research in distributed databases and operating systems is related to the work described here [DGMS85, SKK$^+$90, Kis96, GPJ93, DPS$^+$94, RHR$^+$94, PJG$^+$97, etc.]. The overall goals of all of these systems (especially of distributed filesystems such as Ficus, Bayou, Coda, and LittleWork) are similar to those of Unison. The main differences are that the synchronization operations in these systems are intended to be transparent to the user (they are built into the OS rather than being packages as user-level tools), and—a related but more important point—that they adopt a log-based perspective in which the synchronizer can see a trace of all filesystem activities, not just the final state at the moment of synchronization. Rumor [Rei97] and Reconcile [How99], on the other hand, are user-level synchronizers and share our static approach. Both of these go further than Unison in one significant respect: they seriously address multi-host synchronization, using "gossip architectures." Simpler user-level synchronizers for Unix platforms include Bal [Chr97] and XFiles (`http://www.idiom.com/~zilla/Xfiles/xfiles.html`).

Specifications of synchronizers are much less common than implementations. Other than our own early spec [BP98], we am only aware of Norman Ramsey's "algebraic approach" to synchronization [RC01] (this work was inspired by ours and has similar aims, but uses a log-based approach) and a newer project at Microsoft Research led by Marc Shapiro [SRK00], which also adopts a log-based approach but addresses the more general problem domain of constructing a "reconciliation middleware" platform that can be used by arbitrary application programs.

A different piece of related work is the SyncML standard [Syn] recently proposed by an industrial consortium of computer and PDA manufacturers (including Panasonic, Nokia, Eriksson, Starfish, Moterola, Palm, IBM, Lotus, and Psion). SyncML is more a *protocol* specification than a *system* specification: it is mainly concerned with making sure that the devices engaging in a synchronization operation are speaking the same language, rather than with constraining the overall outcome of the operation. Another major difference is that SyncML deals only with flat record structures (mappings from atomic record-identifiers to records of simple data). Multiple data formats on different devices (related to our cross-platform goals) are supported, but only via application-specific mappings that are regarded as outside the purview of the specification.

10 Conclusions

Our experiences designing and building Unison have taught us many small lessons and a few larger ones. Perhaps the most surprising has been how courageous many people are about trying out little-tested and obviously dangerous tools on their own home directories. Another, less surprising observation has been that, although the principled approach we have adopted has been fairly expensive (we have spent a lot of time at the blackboard that we could have spent at the keyboard), being principled has also led us to many solutions that would probably not have been possible otherwise: it would have been extremely difficult to understand all of the interacting features of the current design without a firm conceptual foundation.

[**Weak:**]Naturally, there is much more work to be done. The extension we are working on hardest at the moment is trying to extend the intuitions gleaned from the Unison effort to the more general problem of synchronizing arbitrary structured data (in the form of XML). Another attractive direction is finding minimally intrusive ways to make a user-level file synchronizer cooperate with the underlying OS to achieve better performance (in particular, much faster update detection). Finally, we are exploring extensions of our design to deal with multiple-replica synchronization.

Acknowledgements

The current version of Unison was designed and implemented by the authors, with substantial contributions by Sylvain Gommier and Matthieu Goulay. Our implementation of the rsync protocol was built by Norman Ramsey and Sylvain Gommier. It is is based on Andrew Tridgell's thesis work and inspired by his rsync utility. Unison's mirroring and merging functionality was implemented by Sylvain Roy. Jacques Garrigue contributed the original Gtk version of the user interface. Heroic alpha-testing by Norman Ramsey, Cedric Fournet, Jaques Garrigue, Karl Crary, and Karl Moerder helped polish rough edges off of early releases. Sundar Balasubramaniam worked with Benjamin Pierce on a prototype implementation of an earlier synchronizer in Java; Insik Shin and Insup Lee contributed design ideas to this implementation. Cedric Fournet contributed to an even earlier prototype.

Comments from Jamey Leifer helped us improve earlier drafts of this paper.

Pierce's work on Unison was partially supported by the NSF under grants CCR-9701826 and 0113226, *ITR/SY+IM: Principles and Practice of Synchronization.* Vouillon's was supported by a fellowship from the University of Pennsylvania's Institute for Research in Cognitive Science (IRCS).

References

[BP98] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, October 1998. Full version available as Indiana University CSCI technical report #507, April 1998.

[Chr97] Jürgen Christoffel. Bal: A tool to synchronize document collections between computers. In *Eleventh Systems Administration Conference (LISA)*, San Diego, CA, October 1997.

[DGMS85] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3), September 1985.

[DPS+94] Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Brent Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, California*, December 1994.

[GPJ93] R. G. Guy, G. J. Popek, and T. W. Page Jr. Consistency algorithms for optimisic replication. In *Proceedings of the First International Conference on Network Protocols*, October 1993.

[How99] John H. Howard. Reconcile user's guide. Technical Report TR99-14, Mitsubishi Electronics Research Lab, 1999.

[Kis96] James Jay Kistler. *Disconnected Operation in a Distributed File System.* PhD thesis, Carnegie Mellon University, 1996.

[KRSD01] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The iccube approach to the reconciliation of divergent replicas. In *Principles of Distributed Computing (PODC)*, 2001.

[PJG+97] T. W. Page, Jr., R. G.. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 11(1), December 1997.

[PV01] Benjamin C. Pierce and Jérôme Vouillon. Unison: A file synchronizer and its specification. Manuscript, 2001.

[RC01] Norman Ramsey and Elöd Csirmaz. An algebraic approach to file synchronization. Submitted for publication, March 2001.

[Rei97] Peter Reiher. Rumor 1.0 User's Manual., 1997. http://fmg-www.cs.ucla.edu/rumor.

[RHR+94] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Conference Proceedings*, June 1994.

[SKK+90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file systems for a distributed workstation environment. *IEEE Transactions on Computers*, C-39(4):447–459, April 1990.

[SRK00] Marc Shapiro, Antony Rowstron, and Anne-Marie Kermarrec. Application-independent reconciliation for nomadic applications. In *Proc. SIGOPS European Workshop: "Beyond the PC: New Challenges for the Operating System"*, Kolding (Denmark), September 2000. ACM SIGOPS. `http://www-sor.inria.fr/~shapiro/papers/ew2000-logmerge.html`.

[Syn] SyncML: The new era in data synchronization. `http://www.syncml.org`.

[TM96] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, 1996.

[Tri99] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.