# What is a File Synchronizer?

S. Balasubramaniam
Vidam Communications
sundar@vidam.com

Benjamin C. Pierce
University of Pennsylvania
bcpierce@cis.upenn.edu

**Abstract**

Mobile computing devices intended for disconnected operation, such as laptops and personal organizers, must employ optimistic replication strategies for user files. Unlike traditional distributed systems, such devices do not attempt to present a "single filesystem" semantics: users are aware that their files are replicated, and that updates to one replica will not be seen in another until some point of synchronization is reached (often under the user's explicit control). A variety of tools, collectively called *file synchronizers*, support this mode of operation.

Unfortunately, present-day synchronizers seldom give the user enough information to predict how they will behave under all circumstances. Simple slogans like "Non-conflicting updates are propagated to other replicas" ignore numerous subtleties—e.g., Precisely what constitutes a conflict between updates in different replicas? What does the synchronizer do if updates conflict? What happens when files are renamed? What if the directory structure is reorganized in one replica?

Our goal is to offer a simple, concrete, and precise framework for describing the behavior of file synchronizers. To this end, we divide the synchronization task into two conceptually distinct phases: *update detection* and *reconciliation*. We discuss each phase in detail and develop a straightforward specification of each. We sketch our own prototype implementation of these specifications and discuss how they apply to some existing synchronization tools.

## 1 Introduction

The growth of mobile computing has brought to fore novel issues in data management, in particular data replication under disconnected operation. Support for replication can be provided either transparently (with filesystem or database support for client-side caching, transaction logs, etc.) or by user-visible tools for explicit replica management. In this paper we investigate one class of user-visible tools—commonly called *file synchronizers*—which allow updates in different replicas to be reconciled at the user's request.

The overall goal of a file synchronizer is easy to state: it must *detect conflicting updates* and *propagate non-conflicting updates*. However, a good synchronizer is quite tricky to implement. Subtle misunderstandings of the semantics of fileystem operations can cause data to be lost or overwritten. Moreover, the concept of "user update" itself is open to varying interpretations, leading to significant differences in the results of synchronization. Unfortunately, the documentation provided for synchronizers typically makes it difficult to get a clear understanding of what they will do under all circumstances: either there is no description at all or else the description is phrased in terms of low-level mechanisms that do not match the user's intuitive view of the filesystem. In view of the serious damage that can be done by a synchronizer with unintended or unexpected behavior, we would like to establish a concise and rigorous framework in which synchronization can be described and discussed, using terms that both users and implementors can understand.

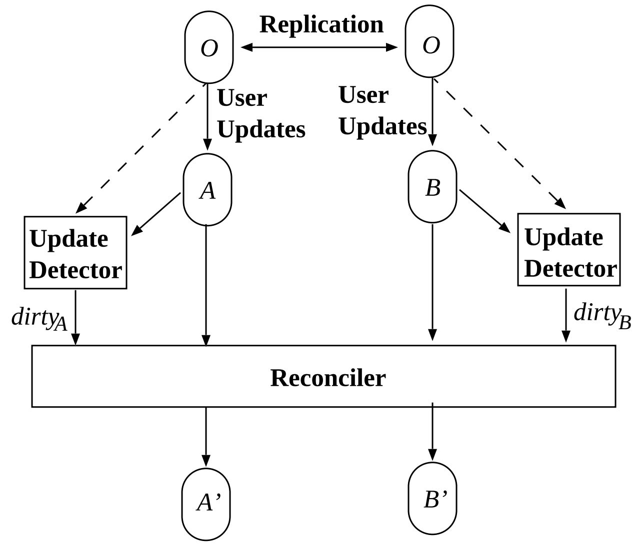
We concentrate on file synchronization in this paper and only briefly touch upon the finer-grained notion of *data synchronization* offered by newer tools [Puma, DDD$^+$94, etc.], but most of the fundamental issues are the same for file and data synchronization. These issues are also closely related to replication and recovery after partitions in mainstream distributed systems [DGMS85, Kis96, GPJ93, DPS$^+$94, etc.]. Ultimately, we may hope to extend our specification to encompass a wider range of replication mechanisms, from data synchronizers to distributed filesystems and databases.

In our model, a file synchronizer is invoked explicitly by an action of the user (issuing a synchronization command, dropping a PDA into a docking cradle, etc.). For purposes of discussion, we identify two cleanly separated phases of the file synchronizer's task: **update detection**—i.e., recognizing where updates have been made to the separate replicas since the last point of synchronization—and **reconciliation**—combining updates to yield the new, synchronized state of each replica.

The update detector for each replica $S$ computes a predicate $dirty_S$ that summarizes the updates that have been made to $S$. (It is allowed to err on the side of safety, indicating possible updates where none have occurred, but all actual updates must be reported.) The reconciler uses these predicates to decide which replica contains the most up-to-date copy of each file or directory. The contract between the two components is expressed by the requirement

$$\begin{array}{l} \text{for all paths } p, \\ \quad \neg \mathit{dirty}_S(p) \\ \quad \Rightarrow \\ \quad \mathit{currentContents}_S(p) = \mathit{originalContents}_S(p), \end{array}$$

which the update detector must guarantee and on which the reconciler relies. The whole synchronization process may then be pictured as follows:




The filesystems in both replicas start out with the same contents $O$. Updates by the user in one or both replicas lead to divergent states $A$ and $B$ at the time when the synchronizer is invoked. The update detectors for the two replicas check the current states of the filesystems (perhaps using some information from $O$ that was stored earlier) and compute update predicates $\mathit{dirty}_A$ and $\mathit{dirty}_B$. The reconciler uses these predicates and the current states $A$ and $B$ to compute new states $A'$ and $B'$, which should coincide unless there were conflicting updates. The specification of the update detector is a relation that must hold between $O$, $A$, and $\mathit{dirty}_A$ and between $O$, $B$, and $\mathit{dirty}_B$; similarly, the behavior of the reconciler is specified as a relation between $A$, $B$, $\mathit{dirty}_A$, $\mathit{dirty}_B$, $A'$, and $B'$.

The remainder of the paper is organized as follows. We start with some preliminary definitions in Section 2. Then, in Sections 3 and 4, we consider update detection and reconciliation in turn. For update detection, we describe several possible implementation strategies with different performance characteristics. For reconciliation, we first develop a very simple, declarative specification: a small set of natural rules that describe the behavior of a typical synchronizer. We then argue that these rules completely characterize the behavior of any synchronizer satisfying them, and finally show how they can be implemented by a straightforward recursive algorithm. Section 5 sketches our own synchronizer implementation, including the design choices we made in our update detector. Section 6 discusses some existing synchronizers and evaluates how accurately they are described by our specification. Section 7 describes some possible extensions.

Most of our development is independent of the features of particular operating systems and the semantics of their filesystem operations; the one exception is in the implementation of update detectors (Section 3.2), which are necessarily system-specific; our discussion there is biased toward Unix. For the sake of brevity, proofs are omitted.

## 2 Basic Definitions

To be rigorous about what a synchronizer does to the filesystems it manipulates, the first thing we need is a precise way of talking about the filesystems themselves.

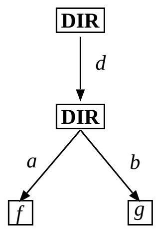
We use the metavariables $x$ and $y$ to range over a set $\mathcal{N}$ of *filenames*. $\mathcal{P}$ is the set of *paths*—finite sequences of names separated by dots. (The dots between path components can be read as slashes by Unix users, backslashes by Windows users, and colons by Mac users.) The metavariables $p$, $q$, and $r$ range over paths. The empty path is written $\epsilon$. The concatenation of paths $p$ and $q$ is written $p.q$. We write $|p|$ for the length of path $p$—i.e., $|\epsilon| = 0$ and $|q.x| = |q| + 1$. We write $q \leq p$ if $q$ is a prefix of $p$, i.e., if $p = q.r$ for some path $r$. We write $q < p$ if $q$ is a *proper* prefix of $p$, i.e., $q \leq p$ and $q \neq p$.

For the purposes of this paper, there is no need to be specific about the contents of individual files. We simply assume that we are given some set $\mathcal{F}$ whose elements are the possible contents of files—for example, $\mathcal{F}$ could be the set of all strings of bytes.

For modeling filesystems, there are many possibilities. Most obviously, we could use the familiar recursive datatype:

$$\mathcal{FS} \quad = \quad \mathcal{F} \uplus (\mathcal{N} \overset{fin}{\rightharpoonup} \mathcal{FS})$$

That is, a "filesystem node" is either a file or a directory, where a file is some $f \in \mathcal{F}$ and a directory is a finite partial function mapping names to nodes of the same form. For example, the filesystem




whose root is a directory containing one subdirectory named $d$, which contains two files $a$ (with contents $f$) and $b$ (with contents $g$), would be represented by the function

$$\begin{array}{l} F = \{d \mapsto D, \\ \qquad n \mapsto \perp \text{ for all other names } n\}, \end{array}$$

where $\perp$ marks positions where $F$ is undefined and $D$ is the function

$$\begin{array}{l} D = \{a \mapsto f,\ b \mapsto g, \\ \qquad n \mapsto \perp \text{ for all other names } n\}. \end{array}$$

For purposes of specification, however, it seems more convenient to use a "flat" representation, where a filesystem is a function mapping whole *paths* to their contents. Formally, we say that a filesystem is an element of the set

$$\begin{array}{rcl} \mathcal{FS} & = & \{\ S \in \mathcal{P} \overset{fin}{\rightharpoonup} (\mathcal{F} \uplus \mathcal{FS}) \ \mid \\ & & \quad \forall p, q \in \mathcal{P}.\ S(p.q) = (S(p))(q)\ \}. \end{array}$$

of finite partial functions from paths to either files or subfilesystems. The constraint on the second line guarantees that we only consider functions corresponding to tree structures—i.e., ones where looking up the contents of a composite path $p.q$ yields the same result as first looking up $p$ and then looking up $q$ in the resulting sub-filesystem (where the application expression $(S(p))(q)$ is defined to yield $\perp$ if $S(p)$ is either $\perp$ or a file).

Under this representation, the example filesystem above corresponds to the function

$$F = \{\epsilon \mapsto F,\ d \mapsto D,\ d.a \mapsto f,\ d.b \mapsto g,\\ p \mapsto \perp \text{ for all other paths } p\},$$

where $D$ is the function

$$D = \{\epsilon \mapsto D,\ a \mapsto f,\ b \mapsto g,\\ p \mapsto \perp \text{ for all other paths } p\}.$$

The metavariables $O$, $S$, $T$, $A$, $B$, $C$, and $D$ range over filesystems.

When $S$ is a filesystem, we write $|S|$ for the length of the longest path $p$ such that $S(p) \neq \perp$. We write $children_A(p)$ for the set of names denoting immediate children of path $p$ in filesystem $A$—that is,

$$children_A(p) = \{q \mid q = p.x \text{ for some } x \ \wedge\ A(q) \neq \perp\}.$$

We write $children_{A,B}(p)$ for $children_A(p) \cup children_B(p)$.

We write $isdir_A(p)$ to mean that $p$ refers to a directory (i.e., not a file and not nothing) in the filesystem $A$. We write $isdir_{A,B}(p)$ iff both $isdir_A(p)$ and $isdir_B(p)$.

To lighten the notation in what follows, we make some simplifying assumptions. First, we assume that, during synchronization, the filesystems are not being modified except by the synchronizer itself. This means that they can be treated as static functions (from paths to contents), as far as the synchronizer is concerned. Second, we assume that, at the end of the previous synchronization, the two filesystems were identical. Third, we handle only two replicas. Finally, we ignore links (both hard and symbolic), file permissions, etc. Section 7 discusses how our development can be refined to relax these restrictions.

## 3 Update Detection

With these basic definitions in hand, we now turn to the synchronization task itself. This section focuses on update detection, leaving reconciliation for Section 4.

### 3.1 Specification

We first recapitulate the specification of the update detector sketched in the introduction:

**3.1.1 Definition:** Suppose $O$ and $S$ are filesystems. Then a predicate $dirty_S$ is said to *(safely) estimate* the updates from $O$ to $S$ if $\neg dirty_S(p)$ implies $O(p) = S(p)$, for all paths $p$.

Among other things, this definition immediately tells us that, if a given path $p$ is not dirty in either replica, then the two replicas have the same contents at $p$.

**3.1.2 Fact:** If $A$, $B$, and $O$ are filesystems and $dirty_A$ and $dirty_B$ estimate the updates from $O$ to $A$ and $O$ to $B$, then $\neg dirty_A(p)$ and $\neg dirty_B(p)$ together imply $A(p) = B(p)$.

One other fact will prove useful in what follows.

**3.1.3 Fact:** For any filesystem $S$, $dirty_S$ is up-closed i.e., if $p \leq q$ and $dirty_S(q)$, then $dirty_S(p)$. We shall use this fact to streamline the specification of reconciliation below.

### 3.2 Implementation Strategies

Update detectors satisfying the above specification can be implemented in many different ways; this section outlines a few and discusses their pragmatic advantages and disadvantages. The discussion is specific to Unix filesystems, but most of the strategies we describe would work with other operating systems too.

#### 3.2.1 Trivial Update Detector

The simplest possible implementation is given by the constantly *true* predicate, which simply marks every file as dirty, with the result that the reconciler must then regard every file (except the ones that happen to be identical in the two filesystems) as a conflict. In some situations, this may actually be an acceptable update detection strategy. On one hand, the fact that the reconciler must actually compare the current contents of all the files in the two filesystems may not be a major issue if the filesystems are small enough and the link between them is fast enough. On the other hand, the fact that all updates lead to conflicts may not be a problem in practice if there are only a few of them. The whole file synchronizer, in this case, degenerates to a kind of recursive remote *diff*.

#### 3.2.2 Exact Update Detector

On the other end of the spectrum is an update detector that computes the *dirty* predicate exactly, for example by keeping a copy of the whole filesystem when it was last synchronized and comparing this state with the current one (i.e., replacing the remote *diff* in the previous case with two local *diff*s).

Detecting updates exactly is expensive, both in terms of disk space and—more importantly—in the time that it takes to compute the difference of the current contents with the saved copies of the filesystem. On the other hand, this strategy may perform well in situations where it is run off-line (in the middle of the night), or where the link between the two computers has very low bandwidth, so that minimizing communication due to false conflicts is critical.

#### 3.2.3 Simple Modtime Update Detector

A much cheaper, but less accurate, update detection strategy involves using the "last modified time" provided by operating systems like Unix. With this strategy, just one value is saved between synchronizations in each replica: the time of the previous synchronization (according to the local clock). To detect updates, each file's last-modified time is compared with this value; if it is older, then the file is not dirty.

Unfortunately, the most naive version of this simple strategy turns out to be wrong. The problem is that, in Unix, renaming a file does not update its modtime, but rather updates the modtime of the directory containing the file: names are a property of directories, not files. For example, suppose we have two files, $a$ and $b$, and that we move $a$ to $b$ (overwriting $b$) in one replica. If we examine just the modtime of the path $b$, we will conclude that it is not dirty, and, in the other replica, $a$ will be deleted without $b$ being changed.

Similarly, it is not enough to look at a file's modtime and its directory's, since the directory itself could have been moved, leaving its modtime alone but changing its parent directory's modtime. To avoid the problem completely, we must judge a file as dirty if *any* of its ancestors (back to the root of the filesystem) has a modtime more recent than the last synchronization. Unfortunately, this makes the simple modtime detector nearly useless in practice, since any update (file creation, etc.) near the root of the tree leads to large subtrees being marked dirty.

#### 3.2.4 Modtime-Inode Update Detector

A better strategy for update detection under Unix relies on both modtimes and inode numbers. We remember not just the last synchronization time, but also the inode number of every file in each replica. The update detector judges a path as dirty if either (1) its inode number is not the same as the stored one or (2) its modtime is later than the last synchronization time. There is no need to look at the modtimes of any containing directories.

For example, if we move $a$ on top of $b$, as above, then the new contents of that replica at the path $b$ will be a file with a different inode number than what was there before. Both $a$ and $b$ will be marked as dirty, leading (correctly) to a delete and an update in the other replica.

We have also experimented with a third variant, where inode numbers are stored only for directories, not for each individual file. This uses much less storage than remembering inode numbers for all files, but is not as accurate. Our own experience indicates that storing all the inode numbers is a better tradeoff, on the whole.

#### 3.2.5 On-Line Update Detector

A different kind of update detector—one that is difficult to implement at user level under Unix but possible under some other operating systems such as Windows—requires the ability to observe the complete trace of actions that the user makes to the filesystem. This detector will judge a file to be modified whenever the user has done anything to it (even if the net effect of the user's actions was to return the file to its original state), so it does not, in general, give the same results as the *exact* update detector. But it will normally get close, and may be cheaper to implement than the exact detector.

On-line upate detection presupposes the ability to track all user actions that affect the filesystem; this places it closer to the domain of traditional distributed filesystems (cf., for example, Coda [Kis96, Kum94], Ficus [RHR+94, PJG+97], Bayou [TTP+95, PST+97], and LittleWorks [HH95]).

## 4 Reconciliation

We now turn our attention to the other major component of the synchronizer, the *reconciler*. We begin by developing a set of simple requirements that any implementation should satisfy (Section 4.1). Then we give a recursive algorithm (Section 4.2) and argue (a) that it satisfies the given requirements, and (b) that the requirements determine its behavior completely, i.e., that any other synchronization algorithm that also satisfies the requirements must be behaviorally indistinguishable from this one (Section 4.3).

### 4.1 Specification

Suppose that $A$ and $B$ are the current states of two filesystems replicating a common directory structure, and that we have calculated predicates $dirty_A$ and $dirty_B$, estimating the updates in $A$ and $B$ since the last time they were synchronized. Running the reconciler with these inputs will yield new filesystem states $C$ and $D$. Informally, the behavioral requirements on the synchronizer can be expressed by a pair of slogans: (1) *propagate all non-conflicting updates*, and (2) *if updates conflict, do nothing.*

(Of course, an actual synchronization tool will typically try to do better than "do nothing" in the face of conflicting updates: it may, for example, apply additional heuristics based on the types of files involved, ask the user for advice, or allow manual editing on the spot. Such cleanup actions can be incorporated in our model by viewing them as if they had occurred just *before* the synchronizer began its real work.)

We are already committed to a particular formalization of the notion of *update* (cf. Section 3): a path is updated in $A$ if its value in $A$ is different from its original value at the time of last synchronization. We can formalize the notion of *conflicting updates* in an equally straightforward way: updates in $A$ and $B$ are conflicting if the contents of $A$ and $B$ resulting from the updates are different. If $A$ and $B$ are both updated but their new contents happen to agree, these updates will be regarded as non-conflicting. (Another alternative would be to say that overlapping updates always conflict. But this will lead to more false positives in conflict detection.)

Our specification of the reconciler can be stated as a set of conditions that should hold between the starting states, $A$ and $B$, and the reconciled states, $C$ and $D$, for every path $p$. Informally:

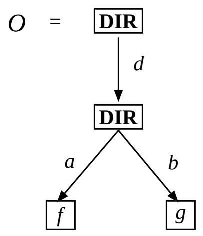
1. If $p$ is not dirty in $A$, then we know that the entire subtree rooted at $p$ has not been changed in $A$, and any updates in the corresponding subtree in $B$ should be propagated to both sides; that is, $C(p)$ (the subtree rooted at $p$ in $C$) and $D(p)$ should be identical to $B(p)$;

2. Conversely, if $p$ is not dirty in $B$, then we should have $C(p) = D(p) = A(p)$.

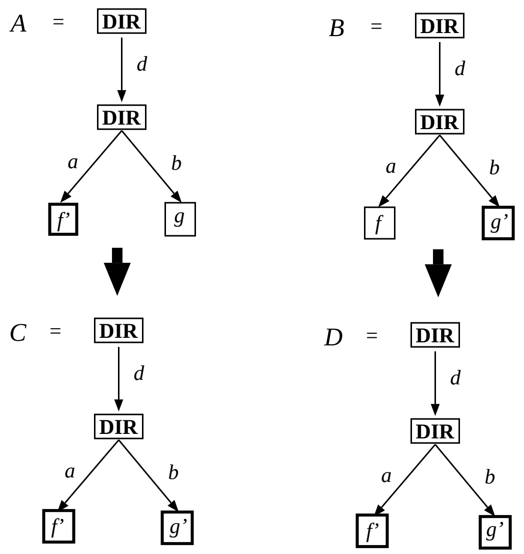
3. If $p$ refers to a directory in both $A$ and $B$, then it should also refer to a directory in $C$ and $D$. (Note that this requirement makes sense whether or not $p$ is dirty in $A$ or $B$.)

4. If $p$ is dirty in both $A$ and $B$ and refers to something other than a directory (i.e., it is either a file or $\perp$) in at least one of $A$ and $B$, then we have potentially conflicting updates. In this case, we should leave things as they are: $C(p) = A(p)$ and $D(p) = B(p)$. (Note that leaving things as they are is the right behavior even in the case where the updates were not actually conflicting—i.e., where it happens that $A(p) = B(p)$.)

A few examples should clarify the consequences of these requirements. Suppose the original state $O$ of the filesystems was

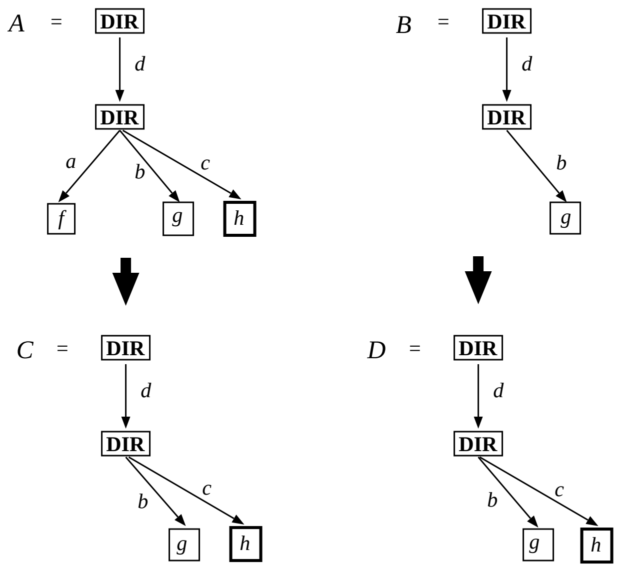


and that we have obtained the current states $A$ and $B$ by modifying the contents of $d.a$ in $A$ and $d.b$ in $B$. Suppose, furthermore (for the sake of simplicity), that we are using an exact update detector, so that $dirty_A$ is *true* for the paths $d.a$, $d$, and $\epsilon$ and *false* otherwise, while $dirty_B$ is *true* for $d.b$, $d$, and $\epsilon$. Then, according to the requirements, the resulting states of the two filesystems should be $C$ and $D$ as shown.



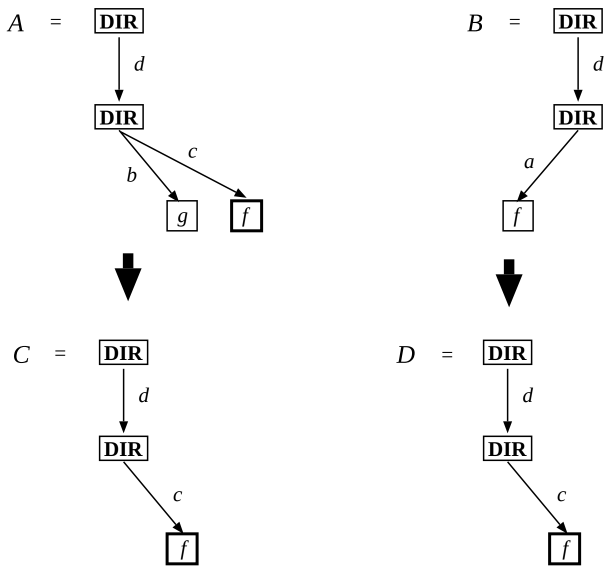

The update in $d.a$ in $A$ has propagated to $B$ and the update in $d.b$ to $A$, making the final states identical.

Suppose, instead, that the new filesystems $A$ and $B$ are obtained from $O$ by adding a file in $A$ and deleting one in $B$:

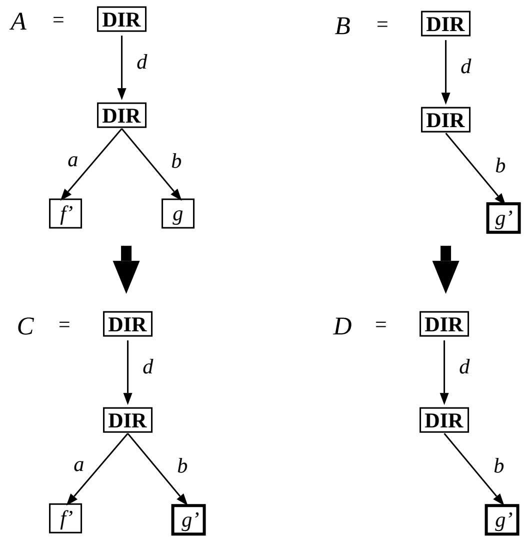



This is an instance of the classic *insert/delete ambiguity* [FM82, GPJ93, PST$^+$97] faced by any synchronization mechanism: if the reconciler could see only the current states $A$ and $B$, there would be no way for it to know that $c$ had been added in $A$, as opposed to having existed on both sides originally and having been deleted from $B$; symmetrically, it could not tell whether $a$ was deleted in $B$ or new in $A$. The *dirty* predicates provided by the update detector resolve the ambiguity: $c$ is dirty only in $A$, while $a$ is dirty only in $B$. (Note that a less accurate update detector might also mark $c$ dirty in $B$ or $a$ dirty in $A$. The effect would then be a conflict reported by the reconciler and no changes to the filesystems—i.e., the specification requires that synchronization "fail safely.")

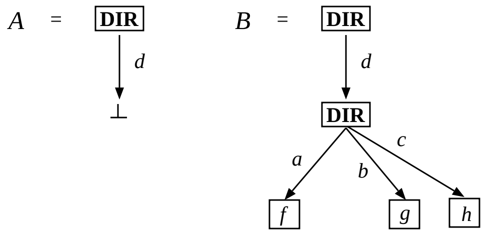
Similarly, suppose the file $d.a$ is renamed, in $A$, to $d.c$, and that $d.b$ is deleted in $B$. In $A$, the paths marked *dirty* are $d.a$, $d.c$, $d$, and $\epsilon$. In $B$, the dirty paths are $d.b$, $d$, and $\epsilon$. So, reconciliation will result in states $C$ and $D$ as shown.




On the other hand, suppose that $d.a$ is modified in $A$ and deleted in $B$, and that $d.b$ is updated only in $B$. The dirty paths in $A$ are $d.a$, $d$, and $\epsilon$; in $B$ they are $d.a$, $d.b$, $d$, and $\epsilon$. The final clause above thus applies to $d.a$, leaving it unmodified in $C$ and $D$, while the update to $d.b$ is propagated to $A$ as usual.



One small refinement is needed to complete the specification of reconciliation. In what we've said so far, we've considered *arbitrary* paths $p$. This is actually slightly too permissive, leading to cases where two of the requirements above make conflicting predictions about the results of synchronization. Suppose, for example, that, $A$ and $B$ are obtained by delete the whole directory $d$ on one side and creating a new file $d.c$ within $d$ on the other:




The contents of $D(d.c)$ should clearly be $h$ after synchronization But what should be the contents of $C(d.c)$? On the one hand, we have $dirty_A(d)$ and $dirty_B(d)$ and $\neg isdir_{A,B}(d)$, so according to the final rule we should have $C(d) = A(d) = \perp$, which implies $C(d.c) = \perp$. But, on the other hand, we have $\neg dirty_A(d.c)$, so according to the first rule, we should have $C(d.c) = B(d.c) = h$.

This is a case of a genuine conflicting update, and we believe the best value for $C(d.c)$ here is $\perp$ (the authors of at least one commercial synchronizer would disagree—cf. Section 6.1). We can resolve the ambiguity by stopping at the first hint of conflict—i.e., by considering only paths $p$ where all the ancestors of $p$ in both $A$ and $B$ refer to directories (and hence do not conflict):

**4.1.1 Definition:** Let $A$ and $B$ be filesystems. A path $p$ is said to be *relevant* in $(A, B)$ iff $\forall q < p \,.\, isdir_{A,B}(q)$.

With this refinement, we are ready to state the formal specification of the reconciler.

**4.1.2 Definition [Requirements]:** The pair of new filesystems $(C, D)$ is said to be a *synchronization* of a pair of original filesystems $(A, B)$ with respect to predicates $dirty_A$ and $dirty_B$ if, for each relevant path $p$ in $(A, B)$, the following conditions are satisfied:

$$\begin{array}{l}
\neg dirty_A(p) \\
\qquad \Longrightarrow \quad C(p) \;=\; D(p) \;=\; B(p) \\
\neg dirty_B(p) \\
\qquad \Longrightarrow \quad C(p) \;=\; D(p) \;=\; A(p) \\
isdir_{A,B}(p) \\
\qquad \Longrightarrow \quad isdir_{C,D}(p) \\
dirty_A(p) \wedge dirty_B(p) \wedge \neg isdir_{A,B}(p) \\
\qquad \Longrightarrow \quad C(p) \;=\; A(p) \;\wedge\; D(p) \;=\; B(p)
\end{array}$$

## 4.2 Algorithm

Having specified the reconciler precisely, we can explore some properties of the specification. In particular, we would like to know that it is *complete*, in the sense that it answers all possible questions about how a reconciler should behave, and that it is *implementable* by a concrete algorithm that terminates on all inputs. We address the latter point first.

For ease of comparison with the abstract requirements above, we present the algorithm in "purely functional" style—as a function taking a pair of filesystems as an argument and returning a fresh pair of filesystems as a result. (Of course, a concrete realization of this algorithm would return no results, performing its task by side-effecting the two filesystems in-place. It should be obvious how to derive such an implementation from the description we give here.)

In the definition, we use the following notation for overwriting part of one filesystem with the contents of the other. Let $S$ and $T$ be functions on paths and $p$ be a path. We write $T \stackrel{p}{\leftarrow} S$ for the function formed by replacing the subtree rooted at $p$ in $T$ with $S$, defined formally as follows:

$$T \stackrel{p}{\leftarrow} S \quad = \quad \lambda q.\ \text{if } p \le q \text{ then } S(q) \text{ else } T(q).$$

**4.2.1 Definition [Reconciliation Algorithm]:** Given predicates $dirty_A$ and $dirty_B$, the algorithm *recon* is defined as follows:

$$\begin{array}{l}
recon(A, B, p) \;=\; \\
\quad 1)\ \text{if } \neg dirty_A(p) \wedge \neg dirty_B(p) \\
\qquad\quad \text{then } (A, B) \\
\quad 2)\ \text{else if } isdir_{A,B}(p) \\
\qquad\quad \text{then let } \{p_1, p_2, \ldots, p_n\} = children_{A,B}(p) \\
\qquad\qquad\qquad \text{(in lexicographic order)} \\
\qquad\quad \text{in let } (A_0, B_0) \;=\; (A, B) \\
\qquad\qquad \text{let } (A_{i+1}, B_{i+1}) \;=\; recon(A_i, B_i, p_{i+1}) \\
\qquad\qquad\quad \text{for } 0 \le i < n \\
\qquad\qquad \text{in } (A_n, B_n) \\
\quad 3)\ \text{else if } \neg dirty_A(p) \\
\qquad\quad \text{then } (A \stackrel{p}{\leftarrow} B, B) \\
\quad 4)\ \text{else if } \neg dirty_B(p) \\
\qquad\quad \text{then } (A, B \stackrel{p}{\leftarrow} A) \\
\quad 5)\ \text{else} \\
\qquad\quad (A, B).
\end{array}$$

That is, *recon* takes a pair of filesystems $A$ and $B$ and a path $p$, and returns a pair of filesystems $(C, D)$ in which the subtrees rooted at $p$ have been synchronized.

An easy induction on $\max(|A|, |B|) - |p|$ shows that *recon* terminates for all filesystems $A$ and $B$ and paths $p$. Also, observe that updates to the filesystems $A$ and $B$ are performed only through the recursive calls and the grafting function defined above; this ensures that $recon(A, B, p)$ leaves unaffected all parts of $A$ and $B$ that are outside the subtree rooted at $p$.

## 4.3 Properties

It remains, now, to verify some properties of the requirements specification and the algorithm. In particular, we can show that (1) the requirements in Definition 4.1.2 fully characterize the behavior of the reconciler; and that (2) the reconciliation algorithm is sound with respect to the specification, i.e., it satisfies the requirements in Definition 4.1.2. It is an immediate consequence of the latter fact that the requirements themselves are consistent, in the sense that, for each $A$, $B$, $dirty_A$, and $dirty_B$, there are some $C$ and $D$ such that $(C, D)$ is a synchronization of $(A, B)$ with respect to $dirty_A$ and $dirty_B$.

To facilitate the correctness arguments, we first introduce a refinement of the original requirements that allows us to focus our attention on a specific region of the two filesystems.

**4.3.1 Definition:** The pair of new filesystems $(C, D)$ is said to be a *synchronization after* $p$ of a pair of original filesystems $(A, B)$ if $p$ is a relevant path in $(A, B)$ and the following conditions are satisfied for each relevant path $p.q$ in $(A, B)$:

$$\begin{array}{l} \neg dirty_A(p.q) \\ \qquad \Longrightarrow \quad C(p.q) = D(p.q) = B(p.q) \\ \neg dirty_B(p.q) \\ \qquad \Longrightarrow \quad C(p.q) = D(p.q) = A(p.q) \\ isdir_{A,B}(p.q) \\ \qquad \Longrightarrow \quad isdir_{C,D}(p.q) \\ dirty_A(p.q) \wedge dirty_B(p.q) \wedge \neg isdir_{A,B}(p.q) \\ \qquad \Longrightarrow \quad C(p.q) = A(p.q) \wedge D(p.q) = B(p.q) \end{array}$$

Note that Definition 4.1.2 is just the special case where $p = \epsilon$.

**4.3.2 Definition:** Paths $p$ and $q$ are *incomparable* if neither is a prefix of the other—i.e., if $p \not\leq q \wedge q \not\leq p$.

**4.3.3 Definition:** We write $sync_p(C, D, A, B)$ if

1. $(C, D)$ is a synchronization of $(A, B)$ after $p$,
2. for all paths $q$, if $p$ and $q$ are incomparable then $C(q) = A(q)$ and $D(q) = B(q)$, and
3. $q \leq p \wedge isdir_{A,B}(q)$ implies $isdir_{C,D}(q)$

The requirements we have placed on the reconciler are *complete* in the sense that they uniquely capture its behavior: given two filesystems which were synchronized at some point in the past, there is at most one pair of new filesystems satisfying the requirements.

**4.3.4 Proposition [Uniqueness]:** Let $A$, $B$, and $O$ be filesystems and suppose that $dirty_A$ and $dirty_B$ estimate the updates from $O$ to $A$ and $B$ respectively. Let $p$ be a relevant path in $(A, B)$. If $(C_1, D_1)$ and $(C_2, D_2)$ are both synchronizations of $(A, B)$ after $p$, then $C_1(p) = C_2(p)$ and $D_1(p) = D_2(p)$.

Furthermore, the requirements are satisfied by the algorithm.

**4.3.5 Proposition [Soundness]:** Let $A$, $B$, and $O$ be filesystems and suppose that $dirty_A$ and $dirty_B$ estimate the updates from $O$ to $A$ and $B$ respectively. Then $recon(A, B, p) = (C, D)$ implies $sync_p(C, D, A, B)$ for any relevant path $p$ in $(A, B)$.

Together, propositions 4.3.5 and 4.3.4 show that algorithm *recon* is actually *equivalent* to the requirements given in Definition 4.1.2. On the one hand, if $(C, D) = recon(A, B, \epsilon)$, then by soundness we know that $(C, D)$ is a synchronization of $A$ and $B$. On the other hand, suppose $(C, D)$ is a synchronization of $A$ and $B$. Since the algorithm is total, it must yield $recon(A, B, \epsilon) = (C', D')$ for some $C'$ and $D'$. But then by uniqueness, we have $C = C'$ and $D = D'$.

## 5 Our Implementation

Our main goal has been to understand the synchronization task clearly, not to produce a full-featured synchronizer ourselves. However, we have found it helpful (as well as useful, for our own day to day mobile computing) to experiment with a prototype implementation that straightforwardly embodies the specification we have described.

Our file synchronizer is written in Java, using Java's *Remote Method Invocation* for networking. The design is intended to perform well over both high- and medium-bandwidth links (e.g., ethernet or PPP). To avoid long startup delays, it uses a modtime-inode strategy (cf. Section 3.2.4) for update detection, requiring only minimal summary information to be stored between synchronizations. It operates entirely at user level, without transaction logs or monitor daemons. It currently handles only two replicas at a time and is targeted toward Unix filesystems (though all but the update detector could be used with any operating system, and new update detection modules should be fairly easy to write).

The user interface (see Figure 1) displays all the files in which updates have occurred, using a tree-browser widget; selecting a file from this tree displays its status in a detail dialog at the right and offers a menu of reconciliation options. In the common case where a file has been updated in only one replica, an appropriate action is selected by default and the tree listing shows an arrow indicating which direction the update will be propagated. If both replicas are updated, the tree view displays a question mark, indicating that the user must make some explicit choice. When the user is satisfied, a single button press fires all the selected actions.

Internally, the implementation closely follows the reconciliation algorithm in Section 4.2 (see Figure 2). At the end of every synchronization, a summary of each replica is stored on the disk. The saved information includes the time when each file in the replica was last synchronized and its inode number at that time. At the beginning of the next synchronization, each update detector reads its summary and traverses the file system to detect updates. A file is marked *dirty* if its *ctime*[1] or inode number has changed since the last synchronization. The reconciler then traverses the two replicas in parallel, examining the files for which updates have been detected on either side and posting appropriate records to a tree of pending actions maintained by the user interface.

## 6 Examples

To explore the utility of our specification, we now discuss some existing synchronizers in terms of the specification framework that we have developed. We do not attempt to provide a complete survey, just a few representative examples.

### 6.1 Briefcase

Microsoft's *Briefcase* synchronizer [Bri98, Sch96] is part of Windows 95/NT. Its fundamental goals seem to match those embodied in our specification ("propagate updates unless they conflict, in which case do nothing by default")—indeed, even its user interface is fairly similar to our prototype. However, some simple experiments revealed several cases where Briefcase's behavior does not match what is predicted by our specification (or any similar specification that we can think of).

[1] In Unix, a file's *ctime* gets changed if the contents or the attributes (such as permission bits) of the file are changed.

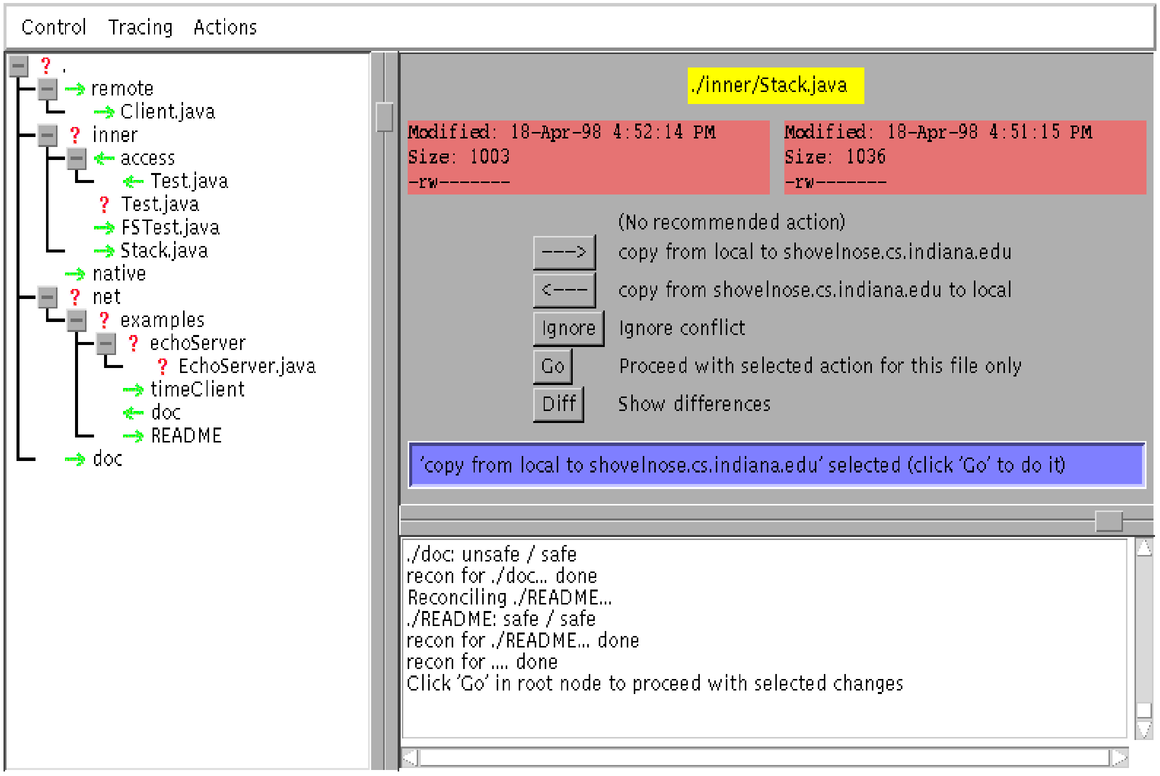



Figure 1: User interface of our synchronizer

The strangest example that we encountered runs as follows. (Since it involves two successive synchronizations, it should be compared with the refined requirements discussed in Section 7.1.) Suppose we have a synchronized filesystem containing a directory (folder) $a$, a subdirectory $a.b$, and a file $a.b.f$. Now, in one replica, we delete $a$ and all its contents; in the other we modify the contents of $a.b.f$ and add a new subdirectory $a.c$; then we synchronize. At this point, Briefcase reports that no updates are needed. (Strictly speaking, this behavior is correct, since it leaves both replicas unchanged, but a conflict should probably have been reported.) Now, in the second replica, we create a new file $a.b.g$, and synchronize again. This time, the synchronizer does propagate some changes: it recreates $a$ in the first replica, adds subdirectories $a.b$ and $a.c$, and copies $a.b.g$—but not $a.b.f$. Success is reported, but the two filesystems are not identical at the end.

### 6.2 PowerMerge

According to the manufacturer's advertising [Pow98], the *PowerMerge* synchronizer from Leader Technologies is "used by virtually every large Macintosh organization and is the highest rated file synchronization program on the market today." We tested the "light" version of the program, which is freely downloadable for evaluation.

Although the description of the program's behavior in the user manual again seems to agree with the intentions embodied in our specification, we were unable to make the program behave as documented. For example, deleting a file on one side and then resynchronizing would lead to the file being re-created, not deleted. Also, when both copies of a file have been modified, the most recent copy is propagated, discarding the update in the other copy.

### 6.3 Rumor

UCLA's Rumor project [Rei97, RPG+96] has built a user-level file synchronizer for Unix filesystems—probably the closest cousin to our own implementation. Although its capabilities go beyond what our specification can describe, Rumor (nearly) satisfies our specification in the two-replica case. (Rumor's model of synchronization originates from the Ficus replicated filesystem; much of our discussion regarding Rumor also applies to the synchronization mechanisms of Ficus [RPG+96, RHR+94, GPJ93].)

In Rumor, reconciliation is performed by a local process in each replica, which works to ensure that the most recent updates to each file in other replicas are eventually reflected in the local state of this replica. For each file in the replica, Rumor maintains a *version vector* reflecting the known updates in all replicas. During reconciliation, this version vector is compared with that of another replica (chosen by the user or determined by availability) to determine which has the latest updates. If the remote copy dominates, then the local copy is modified to reflect the updates; if the local copy dominates, then nothing more is done. (In essence, reconciliation in Rumor uses a "pull model": it is a one-way process.) If there is a conflict, Rumor invokes a *resolver* based on the type of the file; for instance, updates to Unix directories are handled by a "merge resolver" [RHR+94]. Updates eventually get propagated to all replicas by repeated "gossiping" between pairs of replicas.

The update detection strategy in Rumor is a variant of

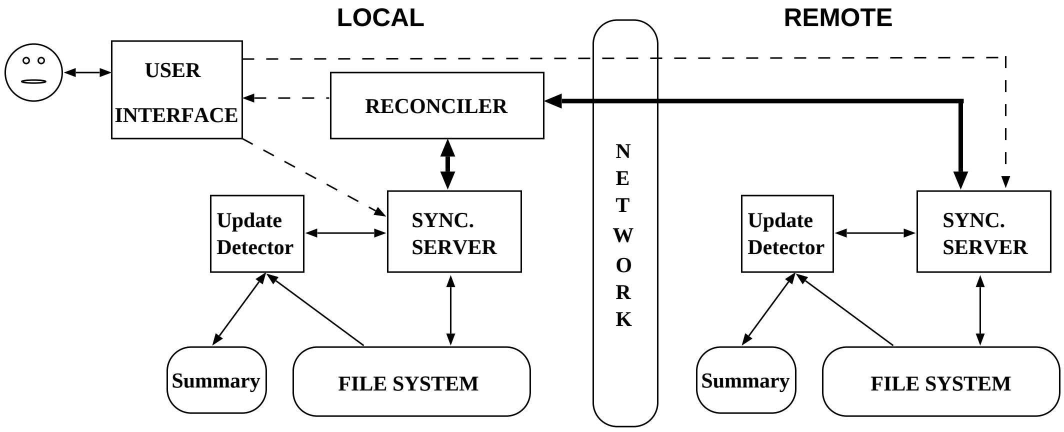



Figure 2: Internals of our synchronizer

the modtime-inode strategy described in Section 3.2.4. Rumor's reconciliation process is more general than that described by our specification. However, it does appear to satisfy our specification if we consider the following special case. (1) There are exactly two Rumor replicas. (2) Both replicas are reconciled at the same time, each treating the other as the source for reconciliation. (3) Overlapping updates are handled by a simple equality check for files (by default, Rumor considers updates to the same file in different replicas as a conflict, even if they result in equal contents) and a recursive merge resolver for directories.

### 6.4 Distributed Filesystems

Not surprisingly, our model of synchronization has some strong similarities to the replication models underlying mainstream distributed filesystems such as Coda [Kis96, Kum94], Ficus [RHR+94, PJG+97], and Bayou [DPS+94, TTP+95]. Related concepts also have a long history in distributed databases (e.g., [Dav84]).

These systems differ from user-level file synchronizers—and from each other—along numerous dimensions... continuous reconciliation vs. discrete points of synchronization, distinguishing or not between client and server machines, eager vs. lazy reconciliation, use of transaction logs vs. immediate update propagation, etc. Since explicit points of synchronization are not part of the user's conceptual model of these systems, our specification framework is not directly applicable. On the other hand, their underlying concepts of optimistic replication and reconciliation are fundamentally very similar to ours. The intention of synchronization—whenever and however it happens—is (eventually) to propagate nonconflicting updates and to detect and repair conflicting updates. Our specification can therefore be viewed as a first step toward a more general framework in which such systems can be described and compared.

One exception is the system described by Mazer and Tardo [MT94]. Their approach is quite similar to ours in that it includes explicit, user-invoked points of synchronization. Apart from the asymmetry in their setting between clients and servers, our framework could be used to model their system.

### 6.5 Data Synchronizers

Much of the engineering effort in commercial synchronization tools goes into facilities for *data synchronization*—merging updates to the same file in different replicas using specific knowledge of the structure of the file based on its type (address book, calendar, etc.). Related approaches have long been pursued in distributed database systems [Dav84]) and has resulted in products like Oracle's Symmetric Replication [DDD+94].

Surprisingly, at least some of these tools can be described very directly in our framework. For example, Puma Technology's popular *Intellisync* [Puma, Pumb] can synchronize many kinds of databases between handheld PDAs, laptop computers, and workstations. It requires that one or more *key fields* be chosen for each type of database to be synchronized. (For example, in an address book the key fields might be the first and last name; in a calendar database they could be the date, time, and description of an appointment.) These key fields correspond to the name of a file in our model. Changing the key fields is like moving the file; changing information in other fields is like changing the contents of the file.

To describe Intellisync in our framework, we just need to generalize the notion of filesystem paths to include names for individual records within files by allowing combinations of key-field values as filename components (e.g., $p = usr.bcp.phonebook.\{lastname{=}Smith, firstname{=}John\}$). The behavior described in the Intellisync manual then follows our specification quite closely. In fact, if we consider the operation of Intellisync just on a single database, then we may drop the clauses of our specification that deal with directories and describe its behavior even more succinctly:

$$
\begin{array}{l}
\neg dirty_A(p) \\
\quad \Longrightarrow \quad C(p) = D(p) = B(p) \\
\neg dirty_B(p) \\
\quad \Longrightarrow \quad C(p) = D(p) = A(p) \\
dirty_A(p) \wedge dirty_B(p) \\
\quad \Longrightarrow \quad C(p) = A(p) \wedge D(p) = B(p)
\end{array}
$$

### 6.6 Version Control Systems

Another class of systems with some striking similarities to file synchronizers is *version control* or *source control* systems like CVS. Such systems include numerous features (version histories, alternative branches, etc.) that fall outside the scope of our specification, but their core behavior includes commands like "check in all changes in this group of files, except in cases where the changes conflict with changes that have already been checked in by another project member." Our requirements might be a useful starting point for full specifications of such systems.

## 7 Extensions

We close by sketching some extensions of our framework.

### 7.1 Partially Successful Synchronization

If it recognizes conflicting updates, the synchronizer may halt without having made the filesystems identical. Then, the next time the synchronizer runs, there will not be one original filesystem, but two. In general, particular regions of the filesystem may have been successfully synchronized at different times. We can easily refine our specification to handle this case. (Our implementation also handles this refinement.)

Instead of assuming that the replicas had some common state $O$ at the end of the previous synchronization, we introduce into the specification a new filesystem $\Gamma$, which records the contents of each path $p$ at the last time when $p$ was successfully synchronized.

The specification of the update detector remains the same as before, except that the *dirty* predicate is defined with respect to $\Gamma$. That is, $dirty_S(p)$ must be *true* whenever $p$ refers in $S$ to something different from what it referred to at the end of the last successful synchronization of $p$.

The reconciler is now extended with an additional output parameter: besides calculating the new states $C$ and $D$ of the two replicas, it returns a new filesystem $\Gamma'$, which will be used as the $\Gamma$ input to the next round of synchronization. For each path $p$, $\Delta(p)$ records the contents of $p$ at the last point where $p$ was successfully synchronized. Formally, we say that the triple $(C, D, \Gamma')$ is said to be a *synchronization* of a pair of original filesystems $(A, B)$ with respect to predicates $dirty_A$ and $dirty_B$ and original state $\Gamma$ if, for each relevant path $p$ in $(A, B)$, the following conditions are satisfied:

$$
\begin{array}{l}
\neg dirty_A(p) \\
\quad \Longrightarrow \quad C(p) = D(p) = B(p) = \Gamma'(p) \\
\neg dirty_B(p) \\
\quad \Longrightarrow \quad C(p) = D(p) = A(p) = \Gamma'(p) \\
isdir_{A,B}(p) \\
\quad \Longrightarrow \quad isdir_{C,D}(p) \ \wedge \ isdir_{\Gamma'}(p) \\
dirty_A(p) \wedge dirty_B(p) \wedge \neg isdir_{A,B}(p) \\
\quad \Longrightarrow \quad C(p) = A(p) \ \wedge \ D(p) = B(p) \\
\qquad\qquad \wedge \text{ if } A(p) = B(p) \ \text{ then } \Gamma'(p) = A(p) \\
\qquad\qquad\qquad\qquad\qquad \text{else } \Gamma'(p) = \Gamma(p)
\end{array}
$$

### 7.2 Multiple Replicas

In general, one may wish to synchronize several replicas on different hosts, not just two. We can generalize our requirements specification to handle multiple replicas in a fairly straightforward way.

Let $Id = \{1, 2, \ldots, n\}$ be a set of tags identifying the $n$ replicas to be synchronized. Let the set of original replicas to be synchronized be denoted by $\mathcal{F}_S = \{S_i \mid i \in Id\}$. For any path $p$, let $D_{p,S}$ be the set of identifiers of replicas that are dirty at $p$—i.e., $D_{p,S} = \{i \mid dirty@_{S_i}(p)\}$. A set of new replicas $\mathcal{F}_R = \{R_i \mid i \in Id\}$ is said to be a *synchronization* of $\mathcal{F}_S$ with respect to dirtiness predicates $dirty@_{S_i}$ if, for each relevant path $p$ in $\mathcal{F}_S$, the following conditions are satisfied:

$$
\begin{array}{l}
D_{p,S} = \emptyset \\
\quad \Longrightarrow \quad \forall i{\in}Id.\ R_i(p) = S_i(p) \\
D_{p,S} \neq \emptyset \ \wedge \ \forall i, j{\in}D_{p,S}.\ S_i(p) = S_j(p) \\
\quad \Longrightarrow \quad \exists j{\in}D_{p,S}.\ \forall i{\in}Id.\ R_i(p) = S_j(p) \\
isdir_S(p) \\
\quad \Longrightarrow \quad isdir_R(p) \\
\exists i, j{\in}D_{p,S}.\ S_i(p) \neq S_j(p) \ \wedge \ \neg isdir_S(p) \\
\quad \Longrightarrow \quad \forall i{\in}Id.\ R_i(p) = S_i(p)
\end{array}
$$

It is interesting to note that Coda's reconciliation strategy depends on a similar requirement. Coda has a certification mechanism which ensures that reconciliation is safe to proceed. Kumar [Kum94, pages 58-61] proves that, if certification succeeds at all servers, then for each data item $d$, either (i) $d$ is not modified in any partition, (ii) the final value of $d$ in each partition is equal to the pre-partition value, or (iii) $d$ was modified in exactly one partition.

In a multi-replica system, the process of reconciliation may in general only involve a subset of the replicas at one time. To describe the intended behavior in this case, we would need to combine the above specification with the refinement described in Section 7.1.

### 7.3 Additional Filesystem Properties

A related generalization offers a natural means of extending our simple model of the filesystem to include properties like read/write/execute permissions, timestamps, type information, symbolic links, etc. For example, a symbolic link can be regarded as a special kind of file whose contents is the target of the link. Similarly, to handle permission bits for files, we take the contents of the file to include both its proper contents and the permission bits.

Hard links are somewhat more difficult to handle, especially if it is possible to create a hard link from inside a synchronized filesystem to some unsynchronized file. However, if this case is excluded, it seems reasonable to handle hard links by annotating each filesystem with a relation describing which files are hard-linked together and taking this additional information into account in the update detector and reconciler.


### Acknowledgments

Marat Fairuzov provided a motivating spark for this work by pointing out some of the subtleties of update detection. Luc Maranget and Peter Reiher gave us the benefit of their own deep experience with writing synchronizers. Jay Kistler and Brian Noble helped explore connections with distributed filesystems and gave us many leads and pointers into the literature in that area. Susan Davidson pointed out useful connections with problems in distributed databases, and Ram Venkatapathy advised us on the mysteries of Windows. Brian Smith contributed his usual boundless enthusiasm and helped us begin to see what it would mean to *really* understand synchronization (in the philosophical sense). Conversations with Peter Buneman, Giorgio Ghelli, Carl Gunter, Bob Harper, Michael Levin, Scott Nettles, and Nik Swoboda helped us improve our presentation of the material. Haruo Hosoya, Michael Levin, Jonathan Sobel, and the MobiCom referees gave us useful comments on earlier drafts of this paper. This work was supported by Indiana University and by NSF grant CCR-9701826.

### References


[Bri98] Microsoft Windows 95: Vision for mobile computing, 1998. http://www.microsoft.com/windows95/info/w95mobile.htm.

[Dav84] S. B. Davidson. Optimism and consistency in partitioned distributed databases. *ACM Transactions on Database Systems*, 9(3), Sep. 1984.

[DDD+94] D. Daniels, L. B. Doo, A. Downing, C. Elsbernd, G. Hallmark, S. Jain, Bob Jenkins, P. Lim, G. Smith, B. Souder, and J. Stamos. Oracle's symmetric replication technology and implications for application design. In *Proceedings of SIGMOD Conference*, 1994.

[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.

[FM82] Michael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982.

[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.

[HH95] L. B. Huston and P. Honeyman. Disconnected Operation for AFS. In *Proceedings of the USENIX Symposium on Mobile and Location Independent Computing*, Spring 1995.

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

[Kum94] Puneet Kumar. *Mitigating the effects of Optimistic Replication in a Distributed File System.* PhD thesis, Carnegie Mellon University, December 1994.

[MT94] Murray S. Mazer and Joseph J. Tardo. A client-side-only approach to disconnected file access. In *Workshop on Mobile Computing Systems and Applications*, December 1994.

[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.

[Pow98] PowerMerge software (Leader Technologies), 1998. http://www.leadertech.com/merge.htm.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16), Saint Malo, France*, October 1997.

[Puma] Designing effective synchronization solutions: A White Paper on Synchronization from Puma Technology. http://www.pumatech.com/syncwp.html.

[Pumb] A white paper on DSX$^{tm}$ Technology – Data Synchronization Extensions from Puma Technology. http://www.pumatech.com/dsxwp.html.

[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.

[RPG+96] P. Reiher, J. Popek, M. Gunter, J. Salomone, and D. Ratner. Peer-to-peer reconciliation based replication for mobile computers. In *European Conference on Object Oriented Programming '96 Second Workshop on Mobility and Replication*, June 1996.

[Sch96] Stu Schwartz. The Briefcase—in brief. *Windows 95 Professional*, May 1996. http://www.cobb.com/w9p/9605/w9p9651.htm.

[TTP+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15), Copper Mountain Resort, Colorado*, December 1995.