What is a File Synchronizer?

S. Balasubramaniam
Indiana University
sbalasub@cs.indiana.edu

Benjamin C. Pierce
Indiana University
pierce@cs.indiana.edu

Indiana University
CSCI Technical Report #507

April 22, 1998

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 filesystems 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-level tools for explicit replica management. In this paper we investigate one class of user-level 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 very 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 relate to.

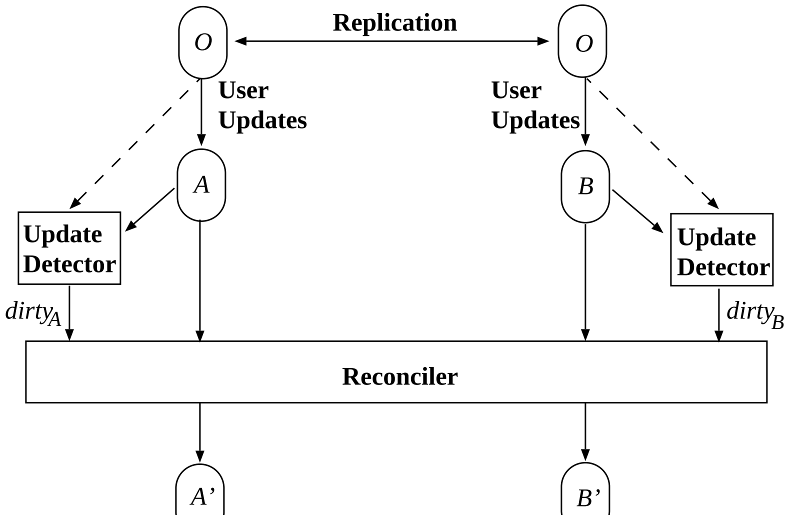
We concentrate on file synchronization in this paper and only briefly touch upon the finer-grained notion of *data synchronization* offered by newer tools. But some of the fundamental issues raised here are relevant for both file and data synchronization. Indeed, these issues are closely related to replication and synchronization (while recovering from a partition) in mainstream distributed systems [DGMS85, Kis96, GPJ93, DPS+94, etc.]. Ultimately, we hope to extend our specification to encompass a wider range of replication mechanisms, from data synchronizers [Puma, DDD+94, etc.] to those of 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 individual filesystem 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

$$\text{for all paths } p, \neg dirty_S(p) \text{ implies } currentContents_S(p) = originalContents_S(p),$$

which the update detector must guarantee and on which the reconciler may rely. 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 $dirty_A$ and $dirty_B$. 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 (or B), and $dirty_A$ (or $dirty_B$); similarly, the behavior of the reconciler is specified as a relation between A, B, $dirty_A$, $dirty_B$, 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 set of four natural rules that any synchronizer should obey. 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, of course, in the implementation of update detectors (Section 3.2), which are necessarily system-specific; our discussion there is biased toward Unix.

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. For most programmers, the first idea that comes to mind is to model filesystems with a simple recursive datatype:

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

That is, a filesystem is either a file or a directory, where a file is some uninterpreted value $f \in \mathcal{F}$ and a directory is a partial function mapping names to filesystems (of the same form).

For present purposes, though, an even simpler and more direct definition will suffice:

$$\mathcal{FS} \quad = \quad \mathcal{P} \rightharpoonup (\mathcal{F} \uplus \{\textsc{dir}\})$$

On this view, a filesystem S is just a partial function mapping each path p to the contents of S at that point—either a file $f \in \mathcal{F}$ or the token DIR, indicating that p names a directory in S. (The contents of this directory do not need to be indicated explicitly, since they can be recovered by evaluating S at extended paths of the form $p.x$.) The advantage of this presentation is that it leads to a "flat" specification, whose properties a user can inspect directly, with no need for inductive reasoning. Of course, formal proofs of these properties will often be inductive, but that need not concern users.

Since "updates" to a filesystem can involve creating and deleting files and directories, as well as modifying existing ones, it is convenient to treat present and absent files uniformly:

$$\mathcal{FS} \quad = \quad \mathcal{P} \to (\mathcal{F} \uplus \{\bot, \textsc{dir}\})$$

In other words, we think of a filesystem S as a *total* function mapping a path p to what p names in S, which can be either a file f, a directory, or nothing at all.

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 some of these restrictions.

2.1 Paths and Files

With the foregoing intuitions in mind, we can now state our basic definitions formally.

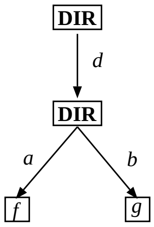
The metavariables x and y range over a set $\mathcal{N}$ of *filenames*. $\mathcal{P}$ is the set of *paths*—finite sequences of names separated by dots. (The dots between path components should be read as slashes by Unix users, backslashes by Windows users, and colons by Mac users.) The metavariables p, q, and r range over paths. The empty path is written ϵ. 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 r.

When f is a function whose domain is the set of paths, we write f/p for the function "f after p", defined as follows: $(f/p)(q) = f(p.q)$.

2.2 Filesystems

For our purposes here, there is no need to be specific about exactly what files can contain. We simply assume that $\mathcal{F}$ is some set whose elements are the possible contents of files—for example, $\mathcal{F}$ could be the set of all strings of bytes. All we need to know about $\mathcal{F}$ is that it supports equality testing: given $f, g \in \mathcal{F}$, we can ask whether $f = g$.

A *filesystem* is a function $S \in \mathcal{P} \to (\mathcal{F} \uplus \{\bot, \textsc{dir}\})$ satisfying the following conditions: (1) if $S(p.x) \neq \bot$, then $S(p) = \textsc{dir}$ ("only directories have children") and (2) there is some depth n such that $S(q) = \bot$ for all paths q of length greater than n ("depth is finite"). For example, the filesystem



in which the root directory contains one subdirectory d, which contains two files a (with contents f) and b (with contents g), is represented by the function

$$\{\epsilon \mapsto \text{DIR},\ d \mapsto \text{DIR},\ d.a \mapsto f,\ d.b \mapsto g, \text{ and } p \mapsto \bot \text{ for all other paths } p\}.$$

The metavariables O, S, 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 $S(p) \neq \bot$. We write $children_A(p)$ for the set of names denoting immediate children of path p in filesystem A—that is,

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

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

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. We first recapitulate the specification of the update detector outlined in the introduction (Section 3.1) and then list several implementation strategies (Section 3.2) that satisfy this specification.

3.1 Specification

We place one additional requirement on the *dirty* predicate computed by the update detector: that it should be upward closed—i.e., that if some path is marked dirty, then so are all of its ancestors. (This condition makes the specification of the reconciler (Definition 4.1.2) slightly simpler.)

3.1.1 Definition: A *dirtiness predicate* is an up-closed predicate on paths, i.e., a predicate ϕ such that, if $p \leq q$ and $\phi(q)$, then $\phi(p)$. An immediate consequence of this definition is that, if ϕ is a dirtiness predicate, then $p \leq q \ \wedge\ \neg\phi(p)$ implies $\neg\phi(q)$. We shall use this fact to streamline the specification of reconciliation below.

3.1.2 Definition: Suppose O and S are filesystems and $dirty_S$ is a dirtiness predicate. Then $dirty_S$ 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.

A crucial property of this definition is that, 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 $A/p = B/p$.

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, this simple strategy turns out to be wrong (under Unix). 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 created.

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) has a modtime more recent than the last synchronization. Unfortunately, this makes the simple modtime detector nearly useless in practice, since an 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 insert 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. Based on experience, our guess is 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. But this presupposes the ability to track arbitrary user actions that affect the filesystem and hence is a preferred strategy for distributed filesystems of various kinds, for instance, 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 show (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 dirtiness 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 utility 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 is to say that overlapping updates always conflict. But this is likely to lead to more false positives in conflict detection.)

There are three ways, then, that synchronization can succeed for a particular path p:

1. if A and B agree at p, then there is nothing to do, and $C(p)$ and $D(p)$ should be the same as $A(p)$ and $B(p)$;

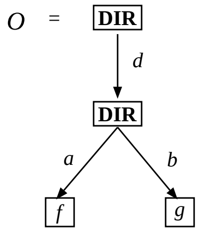
2. if $\neg \mathit{dirty}_A(p)$, then (since dirty_A is up-closed) the entire subtree rooted at p is unchanged in A, and any updates in the corresponding subtree in B should be propagated to both sides; that is both C/p (the subtree rooted at p in C) and D/p should be identical to B/p;

3. conversely, if $\neg \mathit{dirty}_B(p)$, then we should have $C/p = D/p = A/p$.

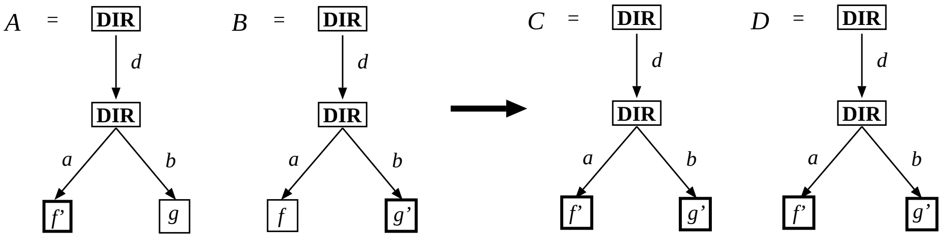
On the other hand, if p has conflicting updates in A and B, we want to leave it unchanged in C and D:

4. if $\mathit{dirty}_A(p)$ and $\mathit{dirty}_B(p)$ and $A(p) \neq B(p)$, then we should have $C/p = A/p$ and $D/p = B/p$.

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

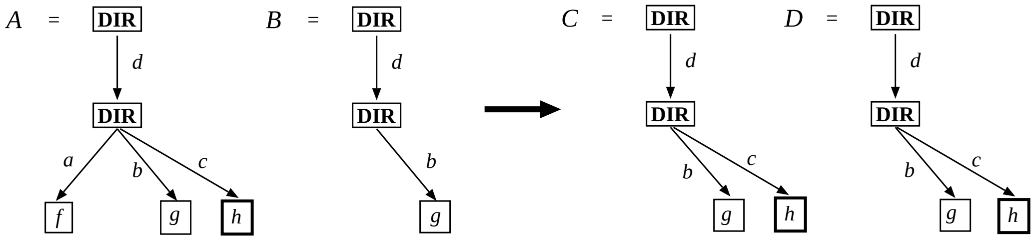


and that we obtain the new states A and B by modifying the contents of $d.a$ in A and $d.b$ in B. Suppose, furthermore (for the sake of simplicity), that we are using an exact update detector, so that $dirty_A$ is *true* for the paths $d.a$, d, and ϵ and *false* otherwise, and $dirty_B$ is *true* for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



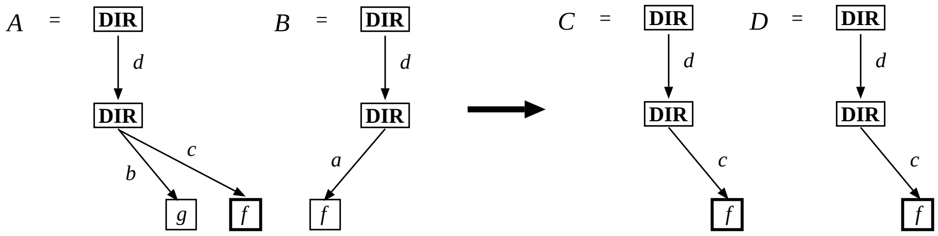
The update in $d.a$ in A has propagated to B and the update in $d.b$ to A, making the final states identical.

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

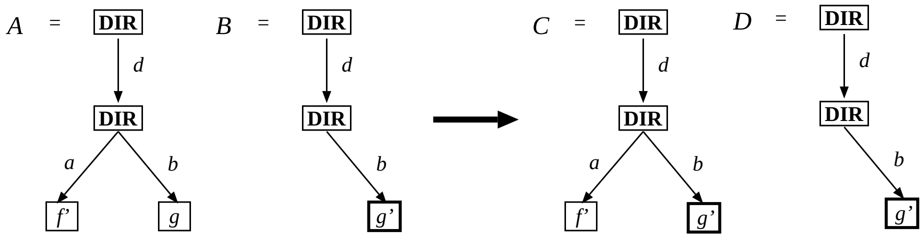


This is an instance of the classic *insert/delete ambiguity* [FM82, GPJ93, PST$^+$97] faced by any synchronization mechanism: if the reconciler could see only the current states A and B, there would be no way for it to know that c had been added in A, as opposed to having been deleted from B (and having existed on both sides originally); symmetrically, it could not tell whether a was deleted in A or new in B. 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 should "fail safely.")

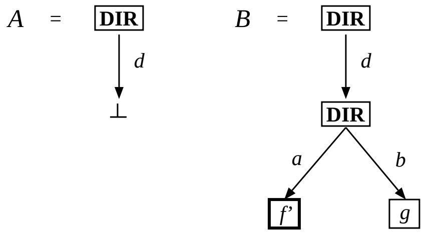
Similarly, suppose the file $d.a$ is renamed, in A, to $d.c$, and that $d.b$ is deleted in B. In A, the paths marked *dirty* are $d.a$, $d.c$, d, and ϵ. In B, the dirty paths are $d.b$, d, and ϵ. So, reconciliation will result in states C and D as shown.



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 ϵ; in B they are $d.a$, $d.b$, d, and ϵ. The fourth 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 that, to obtain A and B, we delete the directory d on one side and modify the file a (within d!) on the other:



What, then, should be the contents of $C(d.b)$? On the one hand, we have $\mathit{dirty}_B(d.b) = \mathit{false}$ (since $O(d.b) = B(d.b) = g$), so according to the third rule we should have $C/d.b = B/d.b$, so $C(d.b) = g$. But, on the other hand, we have both $\mathit{dirty}_A(d)$ and $\mathit{dirty}_B(d)$, and $A(d) \neq B(d)$, so, according to the fourth rule, we should have $C/d = A/d$, from which it follows that $C(d.b) = \bot$.

This is a case of a genuine conflicting update, and we believe the correct value for $C(d.a)$ here is $\bot$ (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 either $p = \epsilon$ or $p = q.x$ for some q and x, with $A(q) = B(q) = \textsc{dir}$.

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 dirtiness predicates dirty_A and dirty_B if, for each relevant path p in (A, B), the following conditions are satisfied:

$$
\begin{array}{lcl}
A(p) = B(p) & \Longrightarrow & C(p) = A(p) \wedge D(p) = B(p) \\
\neg\mathit{dirty}_A(p) & \Longrightarrow & C/p = D/p = B/p \\
\neg\mathit{dirty}_B(p) & \Longrightarrow & C/p = D/p = A/p \\
\mathit{dirty}_A(p) \wedge \mathit{dirty}_B(p) \wedge A(p) \neq B(p) & \Longrightarrow & C/p = A/p \wedge D/p = B/p
\end{array}
$$

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. 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. In the definition, we use the following notation for overwriting part of one filesystem with the contents of the other.

4.2.1 Definition: Let S and T be functions on paths and p be a path. We write $T[p \leftarrow S]$ for the function formed by replacing the subtree rooted at p in T with S, defined formally as follows:

$$T[p \leftarrow S] \quad = \quad \lambda q.\ \text{if } p = q.r \text{ then } S(r) \text{ else } T(q).$$

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

$$
\begin{array}{l}
\mathit{recon}(A, B, p) \;=\\
\quad 1)\ \text{if } \neg\mathit{dirty}_A(p) \wedge \neg\mathit{dirty}_B(p)\\
\qquad\quad \text{then } (A, B)\\
\quad 2)\ \text{else if } A(p) = B(p) = \text{DIR}\\
\qquad\quad \text{then let } \{p_1, p_2, \ldots, p_n\} = \mathit{children}_{A,B}(p) \quad \text{(in lexicographic order)}\\
\qquad\quad \text{in let } (A_0, B_0) \;=\; (A, B)\\
\qquad\qquad \text{let } (A_{i+1}, B_{i+1}) \;=\; \mathit{recon}(A_i, B_i, p_{i+1}) \;\; \text{for } 0 \leq i < n\\
\qquad\qquad \text{in } (An, Bn)\\
\quad 3)\ \text{else if } \neg\mathit{dirty}_A(p)\\
\qquad\quad \text{then } (A[p \leftarrow B/p], B)\\
\quad 4)\ \text{else if } \neg\mathit{dirty}_B(p)\\
\qquad\quad \text{then } (A, B[p \leftarrow A/p])\\
\quad 5)\ \text{else}\\
\qquad\quad (A, B).
\end{array}
$$

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. (Of course, a concrete realization of this algorithm would take only p as argument and 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.)

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 in Definition 4.2.1; this ensures that $\mathit{recon}(A, B, p)$ leaves unaffected all parts of A and B that are outside the subtree rooted at p.

4.3 Soundness and Completeness of the Algorithm

It remains, now, to verify some properties of the requirements specification and the algorithm. In particular, we show that (1) the requirements in Definition 4.1.2 fully characterize the behavior of the reconciler; and that (2) our reconciliation algorithm section is sound with respect to the specification, i.e., it satisfies the requirements in Definition 4.1.2. It is an immediate consequence that the requirements themselves are consistent.

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):

$$
\begin{array}{lll}
RA_1) \;\; A(p.q) = B(p.q) & \Longrightarrow & C(p.q) = A(p.q) \;\wedge\; D(p.q) = B(p.q)\\
RA_2) \;\; \neg\mathit{dirty}_A(p.q) & \Longrightarrow & C/p.q = D/p.q = B/p.q\\
RA_3) \;\; \neg\mathit{dirty}_B(p.q) & \Longrightarrow & C/p.q = D/p.q = A/p.q\\
RA_4) \;\; \mathit{dirty}_A(p.q) \;\wedge\; \mathit{dirty}_B(p.q) \wedge A(p.q) \neq B(p.q) & \Longrightarrow & C/p.q = A/p.q \;\wedge\; D/p.q = B/p.q
\end{array}
$$

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

4.3.2 Definition: We write $synch/_p(C, D, A, B)$ if (1) (C, D) is a synchronization of (A, B) after p, and (2) $p \not\leq q$ implies $C(q) = A(q) \wedge D(q) = B(q)$, for all paths q.

The definition of the reconciler uniquely captures the notion of reconciliation, i.e., given two filesystems which were synchronized at some point in the past, there is at most one pair of new filesystems that satisfies the requirements:

4.3.3 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 $D_1/p = D_2/p$.

Proof: We argue, by induction on $\max(|A|, |B|) - |p.q|$, that

$$C_1/p.q = C_2/p.q \ \wedge\ D_1/p.q = D_2/p.q$$

for any relevant path $p.q$ in (A, B).

There are four cases to consider, depending on whether the path $p.q$ is dirty or not in A and B:

- Suppose $\neg dirty_A(p.q) \ \wedge\ \neg dirty_B(p.q)$.
  Then $\neg dirty_A(p.q.r)$ and $\neg dirty_B(p.q.r)$ for all r, since a dirtiness predicate is up-closed by definition. Then $A(p.q.r) = B(p.q.r)$ for all r, by Definition 3.1.2. So, by (RA_1), we have:

  $$C_1(p.q.r) = A(p.q.r) = B(p.q.r) = D_1(p.q.r) \text{ and}$$
  $$C_2(p.q.r) = A(p.q.r) = B(p.q.r) = D_2(p.q.r).$$

  Thus $C_1/p.q = C_2/p.q$ and $D_1/p.q = D_2/p.q$.

- Suppose $\neg dirty_A(p.q) \ \wedge\ dirty_B(p.q)$.
  Then by (RA_2), we have $C_1/p.q = D_1/p.q = B/p.q$ and $C_2/p.q = D_2/p.q = B/p.q$.
  Thus, $C_1/p.q = C_2/p.q$ and $D_1/p.q = D_2/p.q$.

- Suppose $dirty_A(p.q) \ \wedge\ \neg dirty_B(p.q)$.
  Then we are done, by an argument symmetric to the previous case.

- Finally, suppose $dirty_A(p.q) \ \wedge\ dirty_B(p.q)$.
  There are two cases to consider, depending on whether or not $A(p.q) = B(p.q)$.

  - If $A(p.q) = B(p.q)$ then by clause (RA_1),

    (a) $C_1(p.q) = A(p.q)$ and $D_1(p.q) = B(p.q)$
    (b) $C_2(p.q) = A(p.q)$ and $D_2(p.q) = B(p.q)$.

    Now there are three sub-cases to consider:

    (i) $A(p.q) = B(p.q) \ \in\ \mathcal{F}$
    (ii) $A(p.q) = B(p.q) = \bot$
    (iii) $A(p.q) = B(p.q) = \text{DIR}$

    * In cases (i) and (ii), observations (a) and (b) imply
      $C_1(p.q) = C_2(p.q)$ and $D_1(p.q) = D_2(p.q)$.
      Moreover, by definition of a filesystem,
      $C_1/p.q.r = C_2/p.q.r = D_1/p.q.r = D_2/p.q.r = \bot$
      for all $r \neq \epsilon$. So we have
      $C_1/p.q = C_2/p.q$ and $D_1/p.q = D_2/p.q$,
      as required.

* In case (iii), applying the induction hypothesis yields

  $C_1/p.(q.x) = C_2/p.(q.x)$ and $D_1/p.(q.x) = D_2/p.(q.x)$

  for all x, since each $p.q.x$ is a relevant path. This, along with (a) and (b), implies

  $C_1/p.q = C_2/p.q$ and $D_1/p.q = D_2/p.q$,

  as required.

– If $A(p.q) \neq B(p.q)$ then by clause (RA_4),

$$C_1/p.q = A/p.q \text{ and } D_1/p.q = B/p.q$$

$$C_2/p.q = A/p.q \text{ and } D_2/p.q = B/p.q,$$

which implies $C_1/p.q = C_2/p.q$ and $D_1/p.q = D_2/p.q$, as required. □

The rest of this section leads to a proof of soundness of the algorithm. To prove this, we need to show some more properties of the specification in Section 4.1.2. In particular, we claim

- that the order in which (non-overlapping) regions of a filesystem are synchronized is irrelevant;
- that synchronizing a directory (a region) is just the result of synchronizing its contents (sub-regions).

We begin the argument with a few more definitions.

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

4.3.5 Fact: Let paths p and q be incomparable. If $q \leq r$ then p and r are incomparable.

Proof: This follows easily from the above definition. Assume p and q are incomparable and $q \leq r$. If p and r are not incomparable, then by definition, either $p \leq r$ or $r \leq p$. In either case, we can derive a contradiction:

- If $r \leq p$ then $q \leq p$, which contradicts the assumption p and q are incomparable.
- If $p \leq r$ then either $p \leq q$ or $q \leq p$, which again contradicts the assumption. □

4.3.6 Definition: A set P of paths is said to be *pairwise incomparable* if, for each $p, q \in P$, either $p = q$ or else p and q are incomparable.

4.3.7 Definition: Let P be a set of pairwise incomparable, relevant paths in (A, B). The pair (C, D) is said to be a *synchronization after* P of original filesystems (A, B) if (C, D) is a synchronization after p of (A, B), for each $p \in P$.

4.3.8 Definition: We write $synch/_P(C, D, A, B)$ if:

- (C, D) is a synchronization after P of (A, B)
- for all q, $\neg(\exists p \in P.\ p \leq q)$ implies $C(q) = A(q) \ \wedge \ D(q) = B(q)$.

4.3.9 Lemma [Monotonicity]: Let P be a set of pairwise incomparable and relevant paths in (A, B). Let q be a relevant path in (A, B) such that p and q are incomparable, for any $p \in P$. If $synch/_P(C', D', A, B)$ and $synch/_q(C, D, C', D')$ then $synch/_{P'}(C, D, A, B)$ where $P' = P \cup \{q\}$.

Proof: We prove this in three steps:

1. Let r be any relevant path in (A, B) such that $q \leq r$.
   Then for any such r, $(p \not\leq r)$ for any $p \in P$, by Fact 4.3.5. So $C'(r) = A(r)$ and $D'(r) = B(r)$ by $synch/_P(C', D', A, B)$,and conditions RA_1 through RA_4 of Definition 4.3.1 hold by $synch/_q(C, D, C', D')$. Thus, we conclude that (C, D) is a synchronization after q of (A, B).

2. Let r be any relevant path in (A, B) such that $p \leq r$ for some $p \in P$.
   Then, for any such r, $(q \not\leq r)$ by Fact 4.3.5. So $C(r) = C'(r)$ and $D(r) = D'(r)$ by $\mathit{synch}/_q(C, D, C', D')$, and conditions RA_1 through RA_4 of Definition 4.3.1 hold by $\mathit{synch}/_P(C', D', A, B)$. Thus, we conclude that (C, D) is a synchronization after P of (A, B).

3. Let r be any path such that $(p' \not\leq r)$ for all $p' \in P'$.
   Then $C(r) = C'(r) = A(r)$ and $D(r) = D'(r) = B(r)$ by $\mathit{synch}/_q(C, D, C', D')$ and $\mathit{synch}/_P(C', D', A, B)$

Finally, by (1), (2), and (3), $\mathit{synch}/_{P'}(C, D, A, B)$. □

4.3.10 Lemma [Permutation]: For any n, let $P_n = \{p_i \mid 1 \leq i \leq n\}$ be a set of pairwise incomparable, relevant paths in (A, B). If

1. $(A_0, B_0) = (A, B)$, and
2. $\mathit{synch}/_{p_{i+1}}(A_{i+1}, B_{i+1}, A_i, B_i)$ for all i such that $0 \leq i < n$,

then $\mathit{synch}/_{P_n}(A_n, B_n, A, B)$.

Proof: By induction on n.

- For the base case, where $n = 0$, it is obvious from assumption (1) that $\mathit{synch}/_{\{\}}(A_0, B_0, A, B)$.
- For the induction step, suppose $n \geq 1$. Now, if P_n is pairwise incomparable, then P_{n-1} certainly is. The induction hypothesis thus applies, yielding $\mathit{synch}/_{P_{n-1}}(A_{n-1}, B_{n-1}, A, B))$. Moreover, we have $\mathit{synch}/_{p_n}(A_n, B_n, A_{n-1}, B_{n-1})$. Then by Lemma 4.3.9, $\mathit{synch}/_{P_n}(A_n, B_n, A, B)$. □

4.3.11 Lemma [Children]: Given filesystems A and B, let p be any relevant path in (A, B) such that $A(p) = B(p) = \text{DIR}$ and let $P = \mathit{children}_{A,B}(p)$. Then $\mathit{synch}/_P(C, D, A, B)$ implies $\mathit{synch}/_p(C, D, A, B)$.

Proof: We assume

$$\mathit{synch}/_P(C, D, A, B) \tag{C-1}$$

and show that

1. (C, D) is a synchronization after p of (A, B), and
2. for all q, $(p \not\leq q)$ implies $C(q) = A(q) \ \wedge \ D(q) = B(q)$.

Then $\mathit{synch}/_p(C, D, A, B)$ would follow from (1) and (2).

- To show (2), observe that $(p \not\leq q)$ implies $(p.x \not\leq q)$ for all x. Then by (C-1), $C(q) = A(q)$ and $D(q) = B(q)$.
- To show (1), we show that conditions (RA_1) through (RA_4) of Definition 4.3.1 are satisfied for any relevant path of the form $p.q$. To show these conditions, we consider two sub-cases in turn for each condition, depending on whether or not $q = \epsilon$.
  - Suppose $A(p.q) = B(p.q)$. (A-1)
    - Suppose further that $q = \epsilon$.
      Since $(p.x \not\leq p)$ for all $p.x \in P$, we have $C(p) = A(p)$ and $D(p) = B(p)$ by (C-1) i.e., $C(p.q) = A(p.q)$ and $D(p.q) = B(p.q)$.
    - Suppose instead that $q = x.r$ for some x and r, and let $p' = p.x$.
      Then $p' \in P$ and so $p'.r$ is relevant in (A, B). Also, $A(p'.r) = B(p'.r)$ follows trivially from (A-1). But $\mathit{synch}/_{p'}(C, D, A, B)$ by (C-1). So, by (RA_1), we have $C(p'.r) = A(p'.r)$ and $D(p'.r) = B(p'.r)$, i.e., $C(p.q) = A(p.q)$ and $D(p.q) = B(p.q)$.

    Thus, (RA_1) holds for $p.q$, for all q.
  - Suppose $\neg \mathit{dirty}_A(p.q)$. (A-2)

* Suppose further that $q = \epsilon$.
  Since ($p.x \not\leq p$) for any p.$x \in P$, we have $C(p) = A(p)$ and $D(p) = B(p)$ by (C-1). But by assumption, $A(p) = B(p) = \textsc{dir}$. So

$$C(p) = D(p) = B(p). \tag{C-2}$$

  Also $\neg \mathit{dirty}_A(p.x)$ for all $p.x$, by assumption [A-2] since any dirtiness predicate is up-closed. But $\mathit{synch}/_{p.x}(C, D, A, B)$ for all $p.x \in P$ by (C-1). So by (RA_2), we have

$$C/p.x = D/p.x = B/p.x, \quad \text{for all } p.x \in P. \tag{C-3}$$

  By (C-2) and (C-3), $C/p = D/p = B/p$, i.e., $C/p.q = D/p.q = B/p.q$.
* Suppose instead that $q = x.r$ for some x and r, and let $p' = p.x$;
  Then $p' \in P$ and so $p'.r$ is relevant in (A, B). Then $\mathit{dirty}_A(p'.r)$ follows trivially from (A-2). But $\mathit{synch}/_{p'}(C, D, A, B)$ by (C-1). So by (RA_2), we have C/p'.r = D/p'.r = B/p'.r i.e., $C/p.q = D/p.q = B/p.q$

Thus, (RA_2) holds for p.q for all q.

– Suppose $\neg \mathit{dirty}_B(p.q)$. (A-3)
  Then, (RA_3) holds for $p.q$ for all q, by an argument similar to that for (RA_2).
– Suppose $\mathit{dirty}_A(p.q) \;\wedge\; \mathit{dirty}_B(p.q) \;\wedge\; A(p.q) \neq B(p.q)$. (A-4)
  * Suppose further that $q = \epsilon$.
    But $A(p) \neq B(p)$ contradicts the assumption $A(p) = B(p) = \textsc{dir}$. So q cannot be empty.
  * Suppose instead that $q = x.r$ for some x and r and let $p' = p.x$.
    Then $\mathit{dirty}_A(p'.r) \;\wedge\; \mathit{dirty}_B(p'.r) \;\wedge\; A(p'.r) \neq B(p'.r)$ follows trivially from the assumption. But $\mathit{synch}/_{p'}(C, D, A, B)$ by (C-1). So by (RA_4), we have $C/p'.r = A/p'.r$ and $D(p'.r) = B(p'.r)$ i.e., $C/p.q = A/p.q$ and $D/p.q = B/p.q$

  Thus, (RA_4) holds for $p.q$ for all q. □

We then claim that the algorithm is sound with respect to the requirements specification.

4.3.12 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 $\mathit{recon}(A, B, p) = (C, D)$ implies $\mathit{synch}/_p(C, D, A, B)$ for any relevant path p in (A, B).

Proof: We argue, by induction on $(\max(|A|, B) - |p|)$, that (C, D) is a synchronization of (A, B) after p. Then $\mathit{synch}/_p(C, D, A, B)$ by Definition 4.3.2.

There is one case to consider for each clause of the algorithm. In each case, we show that (C, D) is a synchronization of (A, B) after p.

- **Case 1:** $\neg \mathit{dirty}_A(p) \wedge \neg \mathit{dirty}_B(p)$
  Then $(C, D) = (A, B)$, by algorithm *recon*. We readily verify that conditions (RA_1) through RA_4 of Definition 4.3.1 hold:

  – By Definition 3.1.2, we know $A(p.q) = B(p.q)$ for all q. The premises of clauses (RA_1), (RA_2), and (RA_3) in Definition 4.3.1 now hold for all q, and the corresponding conclusions hold trivially.
  – The premise of clause (RA_4) does not hold for $p.q$ for any q.

  Thus (C, D) is a synchronization of (A, B) after p.

- **Case 2:** $(\mathit{dirty}_A(p) \vee \mathit{dirty}_B(p)) \wedge A(p) = B(p) = \textsc{dir}$
  Then $(C, D) = (A_n, B_n)$ by algorithm *recon*, where if $p_1, p_2, ..., p_n$ is the lexicographic enumeration of $\mathit{children}_{A,B}(p)$, then (An, Bn) is determined by the set of equations:

$$\begin{array}{lcl} (A_0, B_0) & = & (A, B) \\ (A_{i+1}, B_{i+1}) & = & \mathit{recon}(A_i, B_i, p_{i+1}) \end{array}$$

For each $i < n$, by induction hypothesis, (A_{i+1}, B_{i+1}) is a synchronization of (A_i, B_i) after p_{i+1}. So for each $i < n$, $synch/_{p_{i+1}}(A_{i+1}, B_{i+1}, A_i, B_i)$, by Definition 4.3.2. Then $synch/_{children_{A,B}(p)}(A_n, B_n, A, B)$ by Lemma 4.3.10. Finally, by Lemma 4.3.11, $synch/_p(A_n, B_n, A, B)$, as required.

- **Case 3:** $(dirty_A(p) \vee dirty_B(p)) \wedge (A(p) \neq \text{DIR} \vee B(p) \neq \text{DIR}) \wedge \neg dirty_A(p)$
  i.e., $dirty_B(p) \wedge \neg dirty_A(p) \wedge (A(p) \neq \text{DIR} \vee B(p) \neq \text{DIR})$
  Then $(C, D) = (A[p \leftarrow B/p], B)$ by algorithm *recon*.
  We check the clauses of Definition 4.3.1 explicitly, for any arbitrary relevant path $p.q$. Obviously,

  $$D/p.q = B/p.q. \qquad (i)$$

  By Definition 4.2.1,

  $$C/p.q = B/p.q. \qquad (ii)$$

  Also, we have $dirty_A$ since any dirtiness predicate is up-closed,

  $$\neg dirty_A(p.q). \qquad (iii)$$

  Given the above observations, we show that the conditions $(RA1)$ through (RA_4) of Definition 4.3.1 hold for $p.q$:

  – Now, if the premise of clause (RA_1) holds for $p.q$, i.e., if $A(p.q) = B(p.q)$, then by (ii), $C(p.q) = A(.q)$. Along with (i), this ensures that the conclusion of (RA_1) holds for $p.q$, as required.
  – Observation (iii) implies that the premise of clause (RA_2) does hold for $p.q$. But observations (i) and (ii) give $C/p.q = D/p.q = B/p.q$ as required.
  – Next, suppose that the premise of clause (RA_3) holds for $p.q$. Then, for any r with $q \leq r$, we have $\neg dirty_B(p.r)$. By Definition 3.1.2, $A(p.r) = B(p.r)$, and so $A/p.q = B/p.q$ is true. This, along with (i) and (ii), gives us $C/p.q = D/p.q = A/p.q$, as required.
  – Finally, the premise of clause (RA_4) does not hold for $p.q$ for any q, and clause (RA_4) itself is trivially satisfied.

- **Case 4:** $dirty_A(p) \wedge \neg dirty_B(p) \wedge (A(p) \neq \text{DIR} \vee B(p) \neq \text{DIR})$

  The argument for this case is symmetric to the previous case.

- **Case 5:** $dirty_A(p) \wedge dirty_B(p) \wedge (A(p) \neq \text{DIR} \vee B(p) \neq \text{DIR})$
  Then $(C, D) = (A, B)$ by algorithm *recon*.
  If $q \neq \epsilon$, then $p.q$ is not a relevant path. So the only relevant path to be considered is p itself and it is easily verified that conditions (RA_1) through (RA_4) hold for p:

  – If the premise of (RA_1) holds for p, then, since $(C, D) = (A, B)$, we have $C(p) = A(p)$ and $D(p) = B(p)$, as required.
  – Neither of the premises of (RA_2) or (RA_3) hold for p.
  – If the premise of (RA_4) holds for p, then, since $(C, D) = (A, B)$, we have $C/p = A/p$ and $D/p = B/p$, as required. □

4.3.13 Theorem: Algorithm *recon* is sound and complete with the respect to the requirements specification, i.e., if $(C, D) = recon(A, B, \epsilon)$, then (C, D) is a unique synchronization of A and B .

Proof: Propositions 4.3.12 and 4.3.3. □

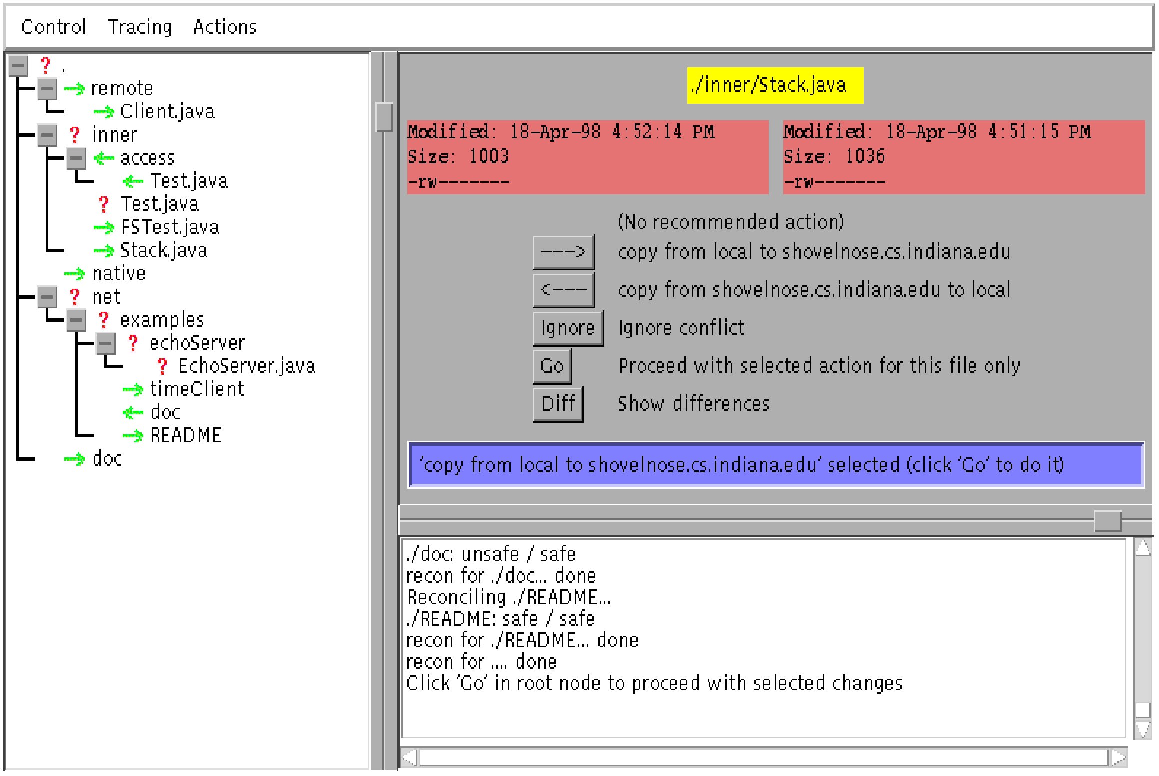


Figure 1: User interface of our synchronizer

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 change 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 keystroke 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 associated with 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

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

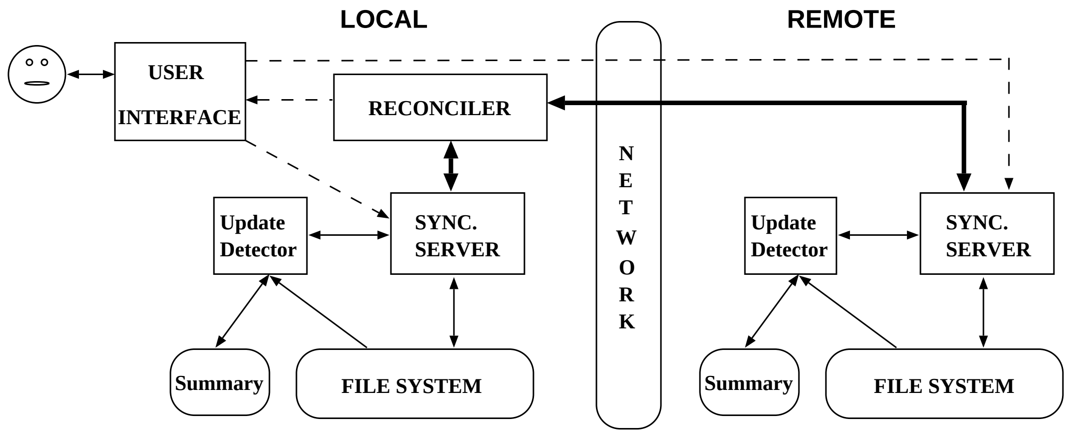


Figure 2: Internals of our synchronizer

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 demonstrate the usefulness of our specification, we now describe 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 ours. However, some simple experiments revealed several cases where Briefcase's behavior does not match what is predicted by our specification.

For example, 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. (This behavior is not really incorrect, since it leaves both replicas unchanged, as demanded by our specification, but a conflict should certainly be reported.) Now, in the second replica, 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 $a.b.f$! This behavior would be judged incorrect by our specification, since it modifies the filesystems after conflicting updates.

This would appear to be a case of a system having been built without a clear specification in mind. The effect is that users are discouraged from trusting the system, since they cannot understand its behavior.

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 is presently able to 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 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 replica 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 uses a combination of the Unix *mtime* and *ctime* file attributes. This strategy appears to satisfy our requirements for update detection in most cases. However, we are not sure about Rumor's behavior in cases like the final example in Section 4.1. Rumor's reconciliation process, on the other hand, 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 striking 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, such as 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. In particular, 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.

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 Γ, 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 Γ. That is, $\mathit{dirty}_S(p)$ 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 predicate *success* that indicates which paths it has succeeded in synchronizing. Formally, we say that the triple $(C, D, \mathit{success})$ is said to be a *synchronization* of a pair of original filesystems (A, B) with respect to dirtiness predicates dirty_A and dirty_B if, for each relevant path p in (A, B), the following conditions are satisfied:

$$\begin{array}{lcl}
A(p) = B(p) & \Longrightarrow & C(p) = A(p) \ \wedge\ D(p) = B(p) \ \wedge\ \mathit{success}(p) \\
\neg\,\mathit{dirty}_A(p) & \Longrightarrow & C/p = D/p = B/p \ \wedge\ \mathit{success}(p) \\
\neg\,\mathit{dirty}_B(p) & \Longrightarrow & C/p = D/p = A/p \ \wedge\ \mathit{success}(p) \\
\mathit{dirty}_A(p) \wedge \mathit{dirty}_B(p) \wedge A(p) \neq B(p) & \Longrightarrow & C/p = A/p \ \wedge\ D/p = B/p \ \wedge\ \neg\mathit{success}(p)
\end{array}$$

The *success* predicate is used after each synchronization step to compute a new "last consistent state", say $\Gamma'(p)$, for each path p:

$$\Gamma'(p) = \begin{cases} C(p) & \text{if } \mathit{success}(p) \\ \Gamma(p) & \text{if } \neg\mathit{success}(p). \end{cases}$$

This filesystem Γ' becomes the Γ that is used by the update detector for the next round of synchronization.

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.

First, we relax the up-closedness condition in our original description of dirtiness predicates: intuitively, we write $\mathit{dirty}@_S(p)$ to mean that S has been updated *exactly at* p, rather than somewhere below p. We say that a dirtiness predicate $\mathit{dirty}@_S$ estimates the updates from O to S if $\neg\mathit{dirty}@_S(p)$ implies $O(p) = S(p)$, for all paths p.

Now, 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., $D_{p,S} = \{i \mid \mathit{dirty}@_{S_i}(p)\}$. 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 $\mathit{dirty}@_{S_i}$ if, for each relevant path p in $\mathcal{F}_S$, the following conditions are satisfied:

$$\begin{array}{lcl}
\forall i, j{\in}Id.\ S_i(p) = S_j(p) & \Longrightarrow & \forall i{\in}Id.\ R_i(p) = S_i(p) \\
D_{p,S} \neq \emptyset \ \wedge\ \forall i, j{\in}D_{p,S}.\ S_i(p) = S_j(p) & \Longrightarrow & \forall i{\in}Id.\ R_i(p) = S_j(p) \ \text{ for some } j{\in}D_{p,S} \\
\exists i, j{\in}D_{p,S}.\ S_i(p) \neq S_j(p) & \Longrightarrow & \forall i{\in}Id.\ R_i(p) = S_i(p)
\end{array}$$

It is interesting to note that Coda's reconciliation strategy depends on a requirement similar to the one above. 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 is modified in exactly one partition. Conditions (i) and (ii) are subsumed by the premise of the first condition of our specification above. The third condition is subsumed by the premise of the second condition of our specification. (The third condition of our specification handles the case when reconciliation fails.)

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 refine the above specification along the lines described in the previous subsection.

7.3 Synchronizing Within Files

Much of the engineering effort in commercial synchronizers(see for instance *Intellisync* [Puma, Pumb]) goes into 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.). This line of research has long been pursued in distributed database systems [Dav84]) and has resulted in products like Oracle's Symmetric Replication [DDD+94]. One way of extending our framework to this additional level of detail would be to generalize the notion of filesystem paths to include names for individual records within files (e.g., $p = \mathit{usr.bcp.phonebook.record13}$ or perhaps even $p = \mathit{usr.bcp.phonebook.}\{\mathit{lastname{=}Smith,firstname{=}John}\}$).

7.4 Additional Filesystem Properties

A related generalization offers a natural means of extending our rather naive picture 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 proper contents and permission bits. This amounts to generalizing our definition of filesystems to

$$\begin{array}{lcl} \mathcal{FS} & = & \mathcal{P} \to \mathcal{F} \\ \mathcal{F} & = & \mathit{Reg}(\textsc{Contents}, \textsc{Perms}) \mid \mathit{Dir}(\textsc{Perms}) \mid \mathit{SymLink}(\textsc{Target}) \mid \bot \end{array}$$

where $\textsc{Contents}$, $\textsc{Perms}$, and $\textsc{Target}$ are some uninterpreted sets.

Recall that the predicate $\mathit{dirty}@_S$ is said to *(safely) estimate* the updates from O to S if $\neg\mathit{dirty}@_S(p)$ implies $O(p) = S(p)$, for all paths p. Also, define $\mathit{dirty}*_S(p)$ as $\forall q\, \neg\mathit{dirty}@_S(p.q)$. With these refinements, we can restate the formal specfication of the reconciler as follows:

7.4.1 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 $\mathit{dirty}@_A$ and $\mathit{dirty}@_B$ if, for each relevant path p in (A, B), the following conditions are satisfied:

$$\begin{array}{lcl} A(p) = B(p) & \Longrightarrow & C(p) = A(p) \wedge D(p) = B(p) \\ \neg\mathit{dirty}*_A(p) & \Longrightarrow & C/p = D/p = B/p \\ \neg\mathit{dirty}*_B(p) & \Longrightarrow & C/p = D/p = A/p \\ \neg\mathit{dirty}@_A(p) \wedge \mathit{dirty}@_B(p) & \Longrightarrow & C(p) = D(p) = B(p) \\ \mathit{dirty}@_A(p) \wedge \neg\mathit{dirty}@_B(p) & \Longrightarrow & C(p) = D(p) = A(p) \\ \mathit{dirty}@_A(p) \wedge \mathit{dirty}@_B(p) & \Longrightarrow & C(p) = A(p) \wedge D(p) = B(p) \end{array}$$

Hard links are 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 (and how they led to misbehavior in an early version of our implementation!). 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, and Jonathan Sobel gave us useful comments on drafts of this paper.

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.

[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.