File Synchronization

Theory and Practice

Benjamin C. Pierce

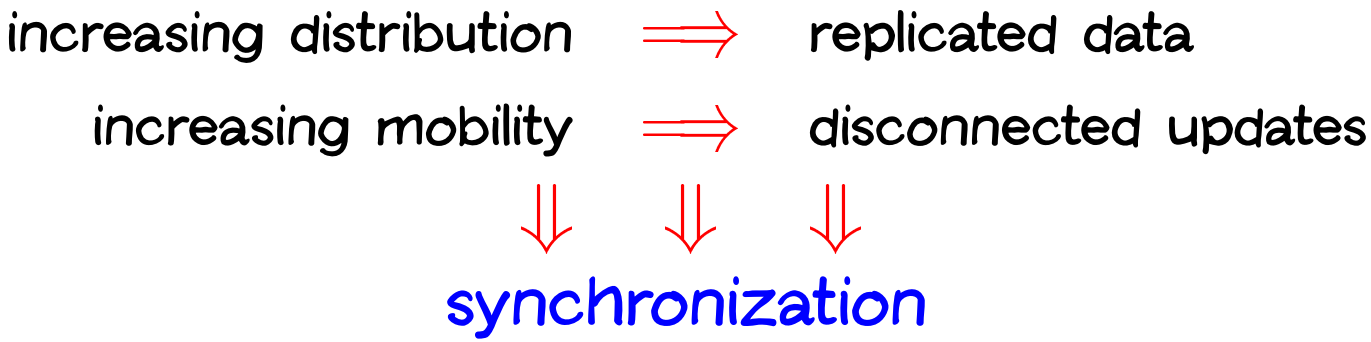
University of Pennsylvania

with

Trevor Jim (AT&T), Jérôme Vouillon (Penn),
Sundar Balasubramaniam (Jareva Tech.), Mattheiu Goulay,
Sylvain Gommier (Ecole Polytechnique)

"First, do no harm..." —Hippocrates

Synchronization all around us...



Examples...

- distributed filesystems and databases (with optimistic replication strategies)
- synchronization utilities for mobile laptops
- "hot sync" software for PDAs
- version control systems (e.g., CVS)
- groupware applications
- ...

The Unison Project

Goals:

Theory: A clean conceptual foundation for file synchronization (and, ultimately, other forms of synchronization).

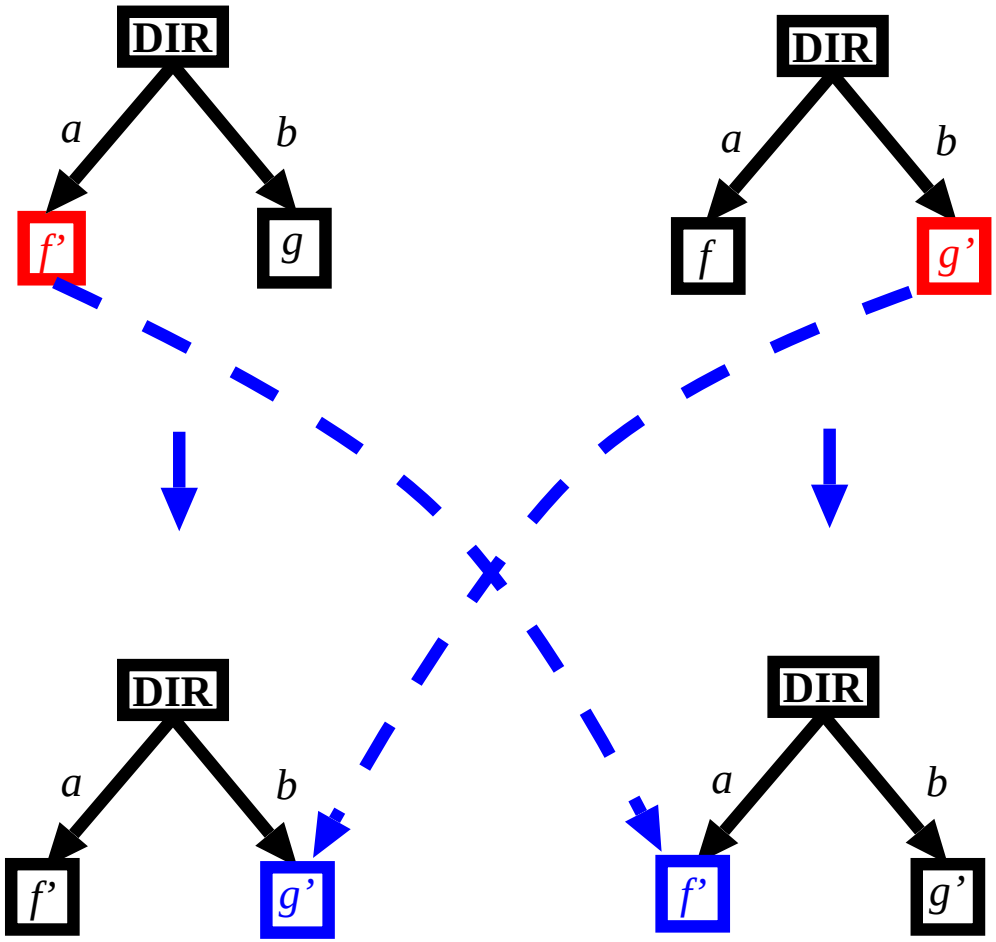
Practice: A robust, portable, cross-platform synchronization tool.

Demo

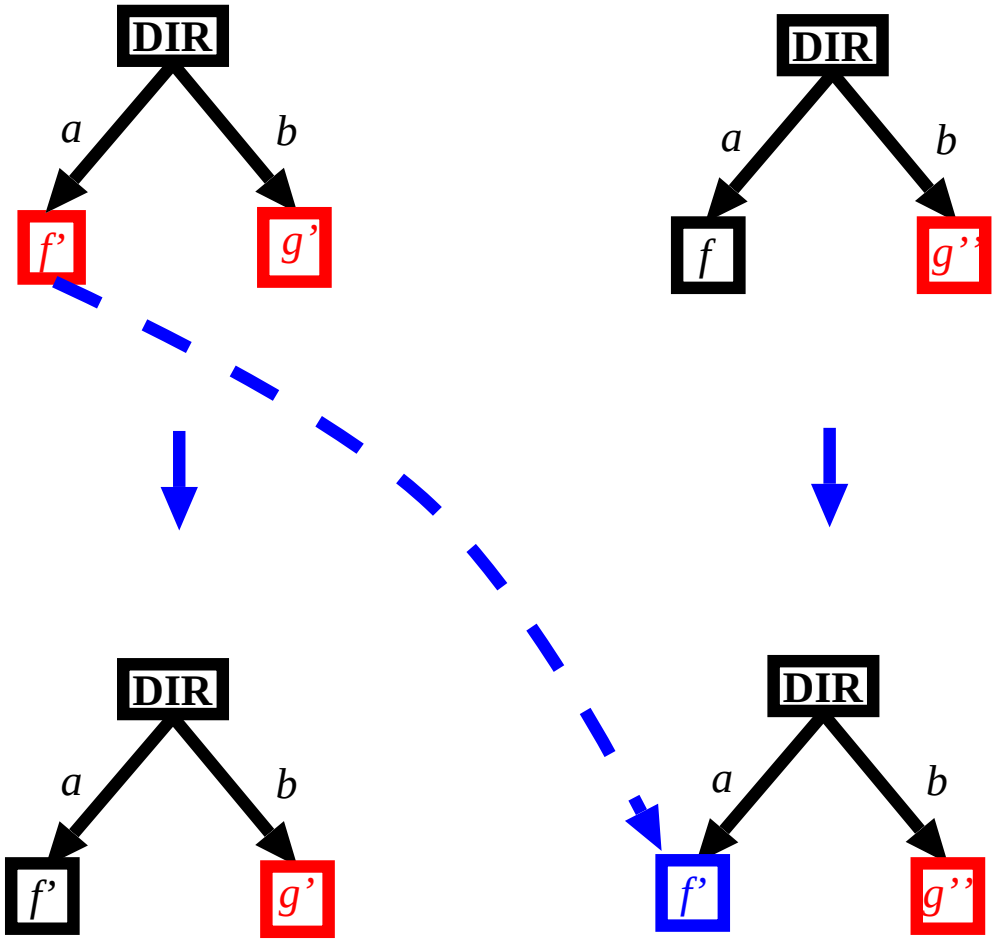
Foundations

Synchronization (a simple example)

A synchronizer should propagate changes...

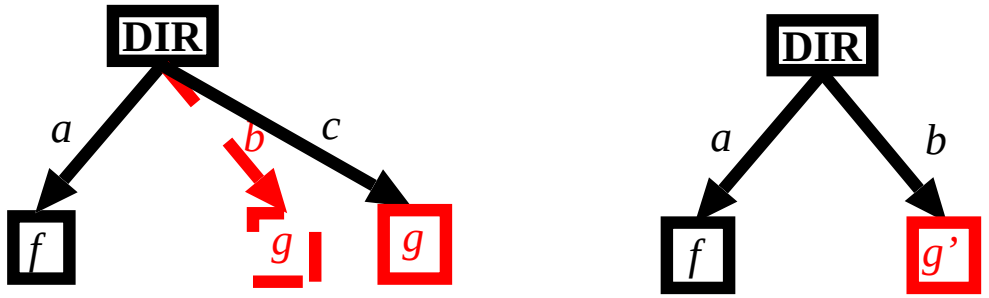


... as long as they do not conflict:



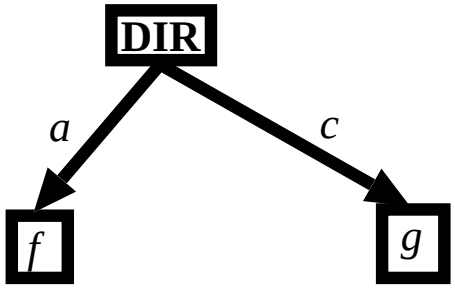
A more interesting example

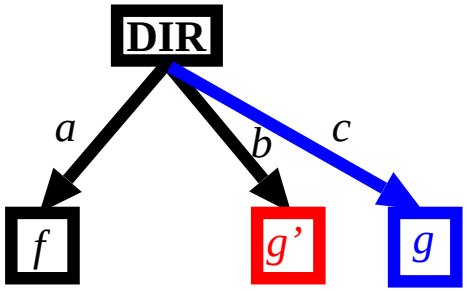
If a file gets renamed on one side and modified on the other, what should the synchronizer do?



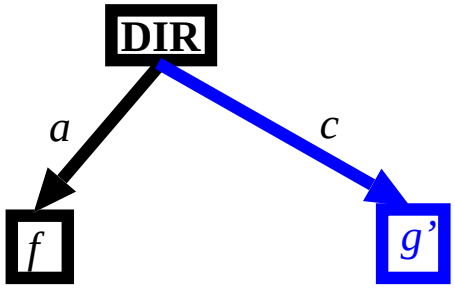
Three reasonable possibilities:

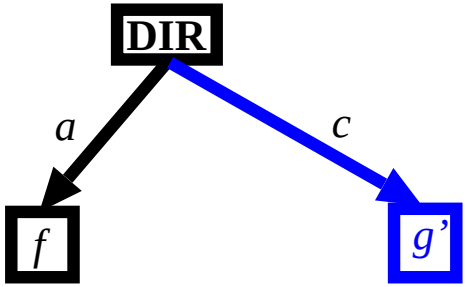
1. Copy old version with new name (c); report a conflict for old name (b)





2. Modify the file in the first replica and move it in the second

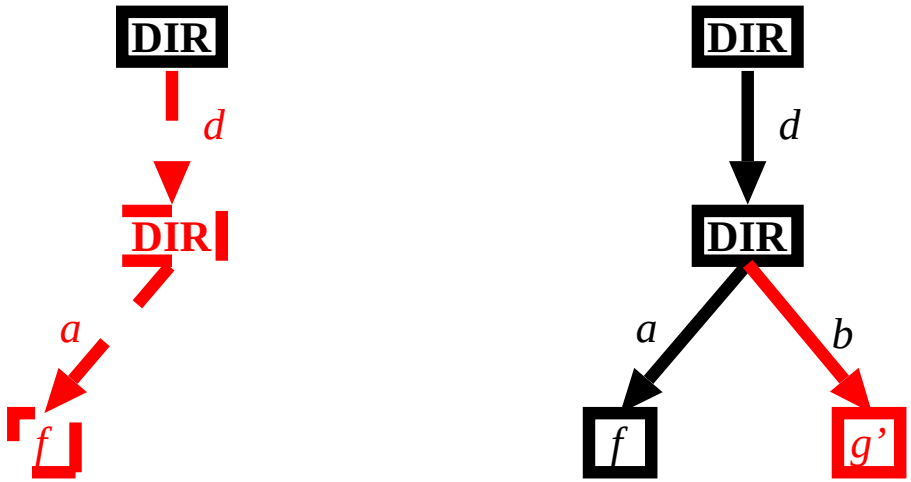




3. Do nothing (report a conflict)

Another unclear case

Suppose a file is created on one side and its parent directory is deleted on the other side...



What should happen?

1. Nothing; a conflict should be reported
2. The siblings ($\mathtt{d.a}$) should be deleted from the second replica, leaving just the file ($\mathtt{d.b}$) and its parent directory ($\mathtt{d}$)
3. The siblings and parent directory should all be deleted from the second replica; the file should be moved to a special "orphanage" and the user alerted

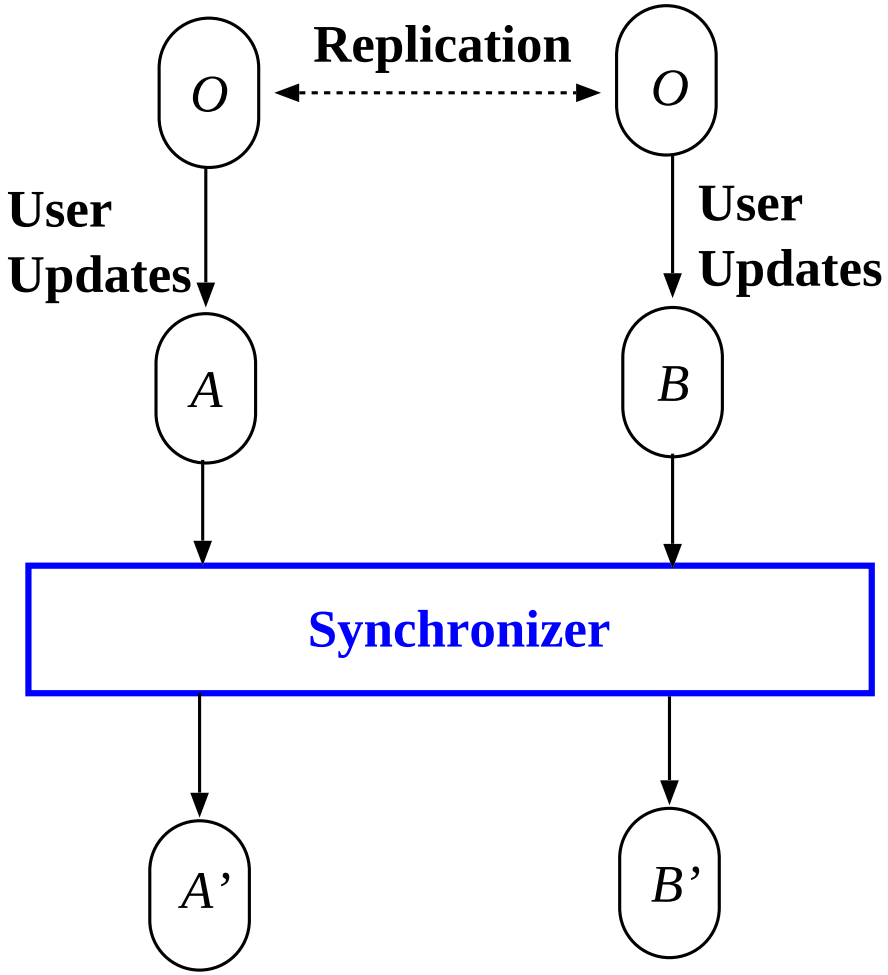
What we want...

A simple, precise framework for specifying and discussing file synchronizers, phrased in terms accessible to both implementers and users.

Organizing principles

- Start by trying to specify just one synchronizer (ours) cleanly
- Specify a user-level synchronizer
  - synchronization operation occurs under explicit user-control
  - only the current state of the filesystems is available to the synchronizer (plus any information it chooses to remember from last time)
- Assume a static model of the world
- Factor out heuristics and user interactions for merging overlapping updates

Structure of the specification



Preliminaries

- A path is a (possibly empty) sequence of names.

  The empty path is written ϵ. The symbol . is used as both the path separator ($\mathrm{d.e.f}$) and for concatenating paths ($\mathrm{p.q}$).

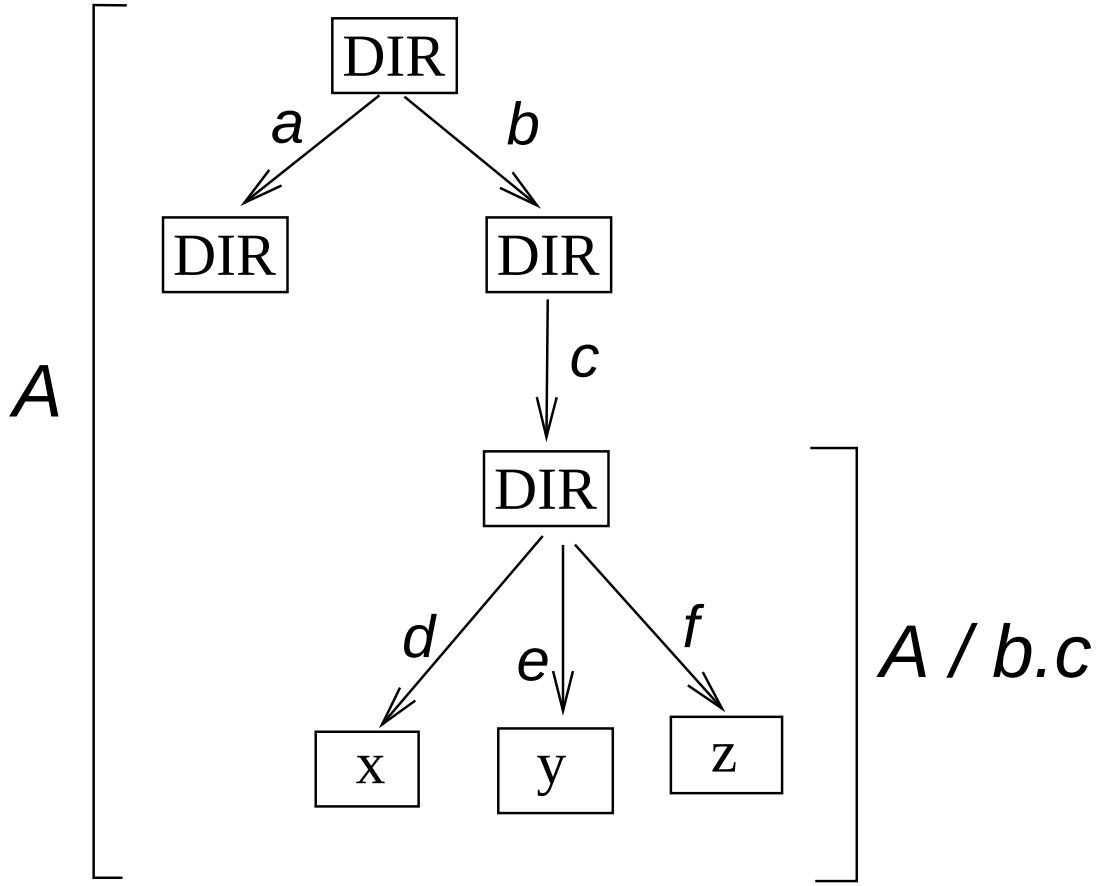
- A file is a value drawn from some uninterpreted set $\mathcal{F}$ (e.g., strings of bytes).

- A filesystem is a total function mapping paths to their contents, where the contents at a path may be a file, a directory, or nothing.

  Formally:

$$\{\ \mathbf{A} \in \mathcal{P} \to (\mathcal{F} \cup \{\mathsf{DIR}, \bot\}) \ \mid$$
$$\forall \mathbf{p}, \mathbf{x}.\ \ \mathbf{A}(\mathbf{p}.\mathbf{x}) \neq \bot \ \Rightarrow\ \mathbf{A}(\mathbf{p}) = \mathsf{DIR}\ \}$$

Write A/p ("A after p") for the sub-filesystem of A rooted at path p



Conflicts

Key question: What is a conflict?

Conflicts

Key question: What is a conflict?

Our answer: A conflict occurs when the two replicas do not agree (at some path), and both have been changed.

Conflicts

Key question: What is a conflict?

Our answer: A conflict occurs when the two replicas do not agree (at some path), and both have been changed.

Formally, we say there is a conflict at path $\mathbf{p}$ if

	$\mathbf{A}(\mathbf{p}) \neq \mathbf{B}(\mathbf{p})$	"$\mathbf{A}$ and $\mathbf{B}$ are different at $\mathbf{p}$"
and	$\mathbf{A}/\mathbf{p} \neq \mathbf{O}/\mathbf{p}$	"$\mathbf{A}$ has been changed at (or below) $\mathbf{p}$"
and	$\mathbf{B}/\mathbf{p} \neq \mathbf{O}/\mathbf{p}$	"$\mathbf{B}$ has been changed at (or below) $\mathbf{p}$"

Core Specification

Each run of a file synchronizer takes filesystems A, B, and O as inputs and yields new filesystems A' and B' as outputs. A run is said to be acceptable if, for all paths p:

(1) if $A(p) \neq O(p)$, then $A'(p) = A(p)$

if $B(p) \neq O(p)$, then $B'(p) = B(p)$

("don't overwrite user changes")

(2) if $A'(p) \neq A(p)$, then $A'/p = B/p$

if $B'(p) \neq B(p)$, then $B'/p = A/p$

("only change replicas by (completely) propagating user changes")

(3) if there is a conflict at p, then $A'/p = A/p$ and $B'/p = B/p$

("don't change (at or below) conflicting paths")

A synchronizer implementation is correct if all its runs are acceptable.

Observations

Interestingly, this specification does not force the synchronizer to do anything at all!

Of course, we prefer that the synchronizer should propagate as many changes as possible, but requiring that it propagate all changes is too strong:

1. The specification should apply even in the case of failure during synchronization.
2. For efficiency, we want to allow the implementation to be conservative in detecting updates — i.e., to give some "false positives," which may lead to false conflicts.

Propagation of updates is thus a nonfunctional requirement: a synchronizer implementation should try to propagate as many changes as it can, subject to the above rules.

Going Deeper

Iterated Synchronization

The synchronizer may fail to make the replicas equal at some path (e.g., because of conflicting changes, or over-conservative change detection).

In this case, what should we use for the "original filesystem" O on the next round of synchronization?

Iterated Synchronization

The synchronizer may fail to make the replicas equal at some path (e.g., because of conflicting changes, or over-conservative change detection).

In this case, what should we use for the "original filesystem" O on the next round of synchronization?

Answer: maintain a (fictitious) filesystem recording the "last synchronized state" of each path.

$$O'(p) = \begin{cases} A'(p) & \text{if } A'(p) = B'(p) \\ O(p) & \text{otherwise} \end{cases}$$

Strictly speaking...

To deal with the possibility of machine failures during synchronization, we need to treat O' as an output of the synchronizer. The specification is extended so that the following values for O' are considered to be correct:

1. $O' = O$
   unchanged (failure during update detection)
2. $O'(p) = \begin{cases} A(p) & \text{if } A(p) = B(p) \\ O(p) & \text{otherwise} \end{cases}$
   recording just the paths already synchronized in the inputs (failure during change propagation)
3. $O'(p) = \begin{cases} A'(p) & \text{if } A'(p) = B'(p) \\ O(p) & \text{otherwise} \end{cases}$
   additionally recording the paths that have just become synchronized (successful termination)

Synchronizing Multiple replicas

We've treated just the two-replica case in this specification (and in our implementation).

Pairwise synchronization can be used to keep 3-5 replicas in sync. Just synchronize successive pairs in a star or ring topology.

For synchronizing more replicas, both specification and implementation can be extended straightforwardly... iff we require that all replicas participate in every synchronization.

For synchronizing many replicas, we need to deal with the fact that only a subset may participate in any given sync. Problems become significantly trickier. (Need something like version vectors.)

Dealing with Links

Synchronization of Unix-style symbolic links can easily be handled in our framework. A symbolic link is just a special kind of file whose "contents" is a string. Both the "ordinary file / symlink" bit and the link-target string are considered part of the contents of the file as far as the synchronizer is concerned.

(Hard links are more problematic.)

Permission bits...

Handled just like symlinks: we consider them as part of the "contents" of the file.

Heterogeneity

Unison is the only synchronizer (AFAWK) that tries to do a good job of synchronizing across different filesystem architectures (Win32 / Posix).

This involves dealing with...

- different permission bits
- different modtime representations
- file name capitalization
- UID/GIDs (between different Unix systems)
- etc.

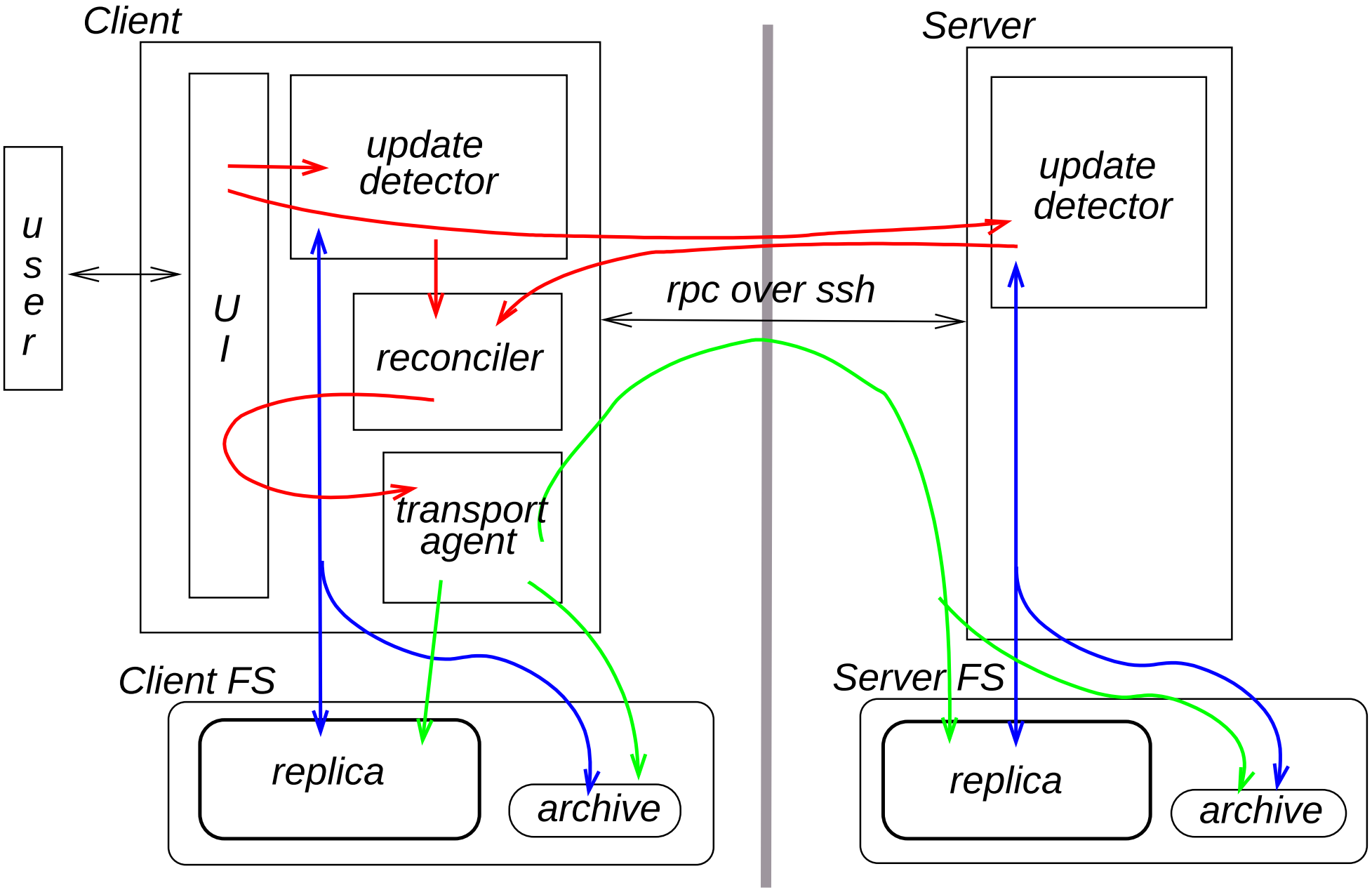
To achieve this, we need to change our goal to "synchronizing the common information" (and doing something reasonable with the rest).

Implementation

Unison

The Unison synchronizer aims for robustness, portability, and heterogeneity...

- Design strongly influenced by the specification described earlier, and vice versa
- Runs on Windows [98/NT/2K] and most flavors of Unix
- Supports cross-platform synchronization between Windows and Unix
- Deals with symlinks, file permissions, modtimes, uids, etc., etc.
- Tuned for high- (ethernet) and medium-bandwidth (PPP) connections
- Uses the rsync protocol for "diffs only" transmission of small updates to large files
- Tunnels over ssh for security (can also use raw sockets)
- Easy install (single executable, no administrative privileges required)
- Source code available under GPL (~15K lines of O'Caml)
- Growing user community (~500-1000 users, max replicas ~5 Gb)



System Architecture

Robustness

Our promise* to users:

> After any run of Unison (whether successful or not), each path in each replica will be either unchanged, or (if permitted by the specification) updated to exactly match the other replica.

Issues:

- Safety for arbitrary crash failures
- Atomicity of changes to filesystems
- Resilience to concurrent activity by the user
- etc.

*modulo bugs (natch), plus a few unavoidable races

Going further

("what do you want to synchronize today...?")

Data synchronization

Many commercial synchronization tools are able to synchronize individual records within databases. For each database, certain fields are designated as key fields. Two records are regarded as "the same record" if they have identical key fields.

Our framework incorporates this case without change. We just have to extend the notion of "path" to include the key fields.

E.g., suppose the path $\mathrm{d.f}$ refers to a database

First Name	Last Name	Age	Address
Adam	Smith	275	Scotland
John	Keynes	115	England
...	...	...	...

and that the key fields of this database are First Name and Last Name.

Then the path $\mathrm{d.f.}\langle \mathrm{Adam}, \mathrm{Smith} \rangle$ refers to the record with contents $\langle 275, \mathrm{Scotland} \rangle$.

XML Synchronization

Key issue:

There are many ways to index information in XML structures (hence, it is not clear how to "match up" the parts of the structures should be synchronized).

E.g., in

```
<table>
  <header>  <column>Names</column> <column>Phones</column>    </header>
  <element>   <name>Joe</name>     <phone>123-4567</phone>    </element>
  <element>   <name>Amy</name>     <phone>888-2001</phone>    </element>
  <element>   <name>Fred</name>    <phone>777-1234</phone>    </element>
  <element>   <name>Eve</name>     <phone>932-3528</phone>    </element>
</table>
```

is the "path" to 777-1234...

- the second child of the fourth child of the root?
- the `phone` child of the third `element` child of the root?
- the `phone` child of the `element` whose `name` is `Fred`?

All are plausible!

How can we figure out what indexing method is intended?

How can we figure out what indexing method is intended?

1. guess

How can we figure out what indexing method is intended?

1. guess
2. ask the user

How can we figure out what indexing method is intended?

1. guess
2. ask the user
3. look at the schema!

Another issue: Ordering

Although the absolute position of a piece of information is generally not its primary index, it is often desirable to maintain ordering of children.

In effect, there can be multiple relevant indexing schemes for a given part of an XML document.

Finishing up...

Related Projects

Lots of implementations:

- Many distributed file systems [Coda, Bayou, Ficus, etc., etc.]
- Many commercial products for Windows and MacOS [MS Briefcase, Puma IntelliSync, etc.]
- Rumor [UCLA]
- Reconcile [Mitsubishi Research]

A few specifications:

- Norman Ramsey [Harvard]

  algebraic specifications of unison-like synchronizers
- Marc Shapiro & co [MSR-UK]

  trace-based specifications of more general middleware layers

Want to play?...

http://www.cis.upenn.edu/~bcpierce/unison

Extra slides...

Examples

We tested some popular synchronizers to see whether they satisfy our specification...

- Microsoft Briefcase...

  Yes, modulo some bugs and differences in intended behavior

- PowerMerge (Mac)...

  No

- Rumor (a Unix-only synchronizer from UCLA)...

  Pretty much (extra generality of Rumor makes comparison hard)

- Distributed filesystems (CODA, Ficus, Bayou, etc.)...

  Pretty much (again, modulo extra generality)

- "Data synchronizers" (Intellisync, etc.)...

  Yes

Strategies for update detection

- Exact update detector

  $\mathrm{dirty}(p)$ iff $\langle$current contents at $p\rangle \neq O(p)$

- Modtime update detector (for Unix)

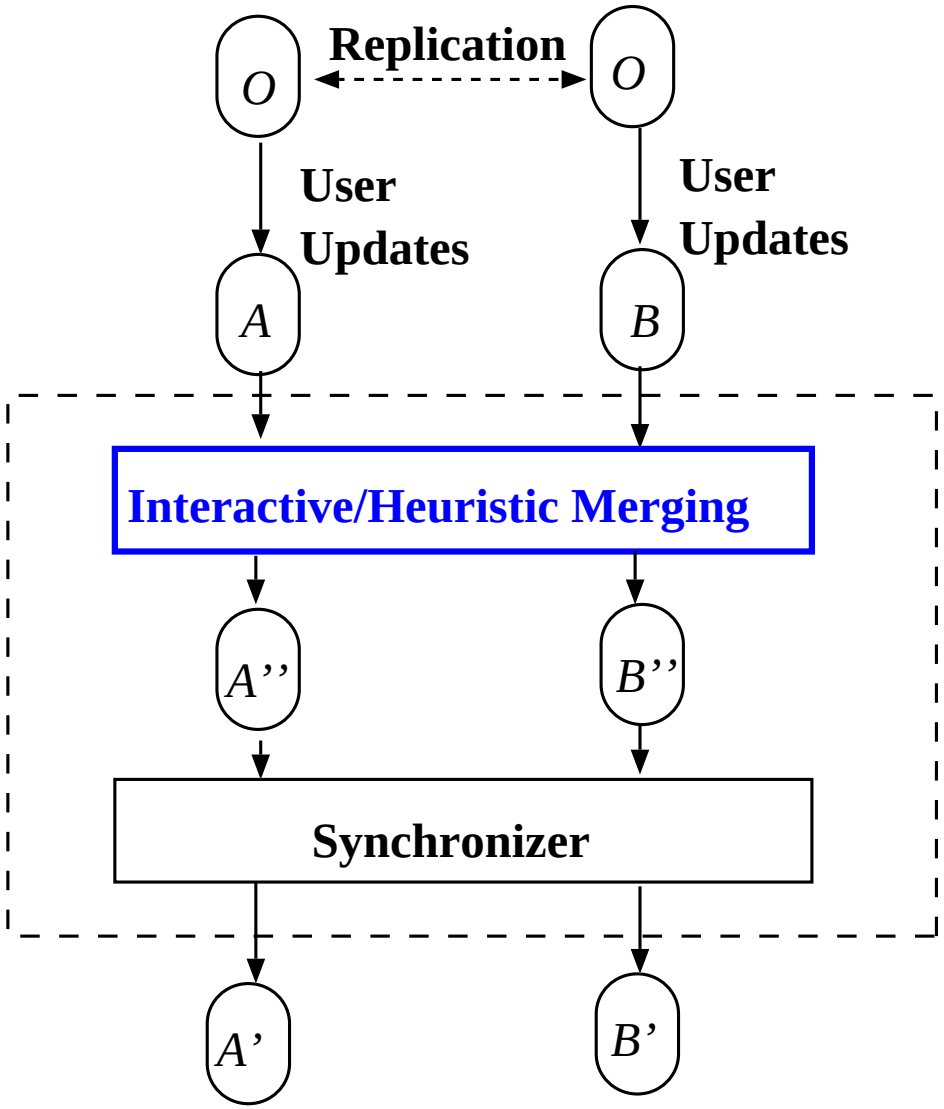
  $\mathrm{dirty}(p)$ iff for some ancestor q of p,

  $\mathrm{modtime}(q) > \langle$last sync time for $q\rangle$

  [Note that $\mathrm{dirty}(p)$ iff $\mathrm{modtime}(p) > \langle$last sync time for $p\rangle$ is not right!]

- Modtime-inode update detector (for Unix)

  $\mathrm{dirty}(p)$ iff $\mathrm{modtime}(p) > \langle$last sync time for $p\rangle$

  or $\mathrm{inode}(p) \neq \mathrm{inode}_O(p)$

Incorporating heuristic / interactive merging