What is a File Synchronizer?

Benjamin C. Pierce

University of Pennsylvania

with

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

"First, do no harm..." — Hippocrates

Synchronizers are everywhere...

increasing distribution $\rightarrow$ replicated data
increasing mobility $\rightarrow$ disconnected updates
$\Downarrow$
need for synchronization tools

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

… but what, exactly, are they doing?

A synchronizer is a very intimate tool:

- incorrect behavior is disastrous
- unexpected behavior can also be pretty bad.

However, users are seldom given enough information to predict a synchronizer's behavior with confidence. Descriptions are either

- high-level, but too vague (users' manuals), or
- precise, but too low-level (Ph.D. theses).

The Unison project

Goals:

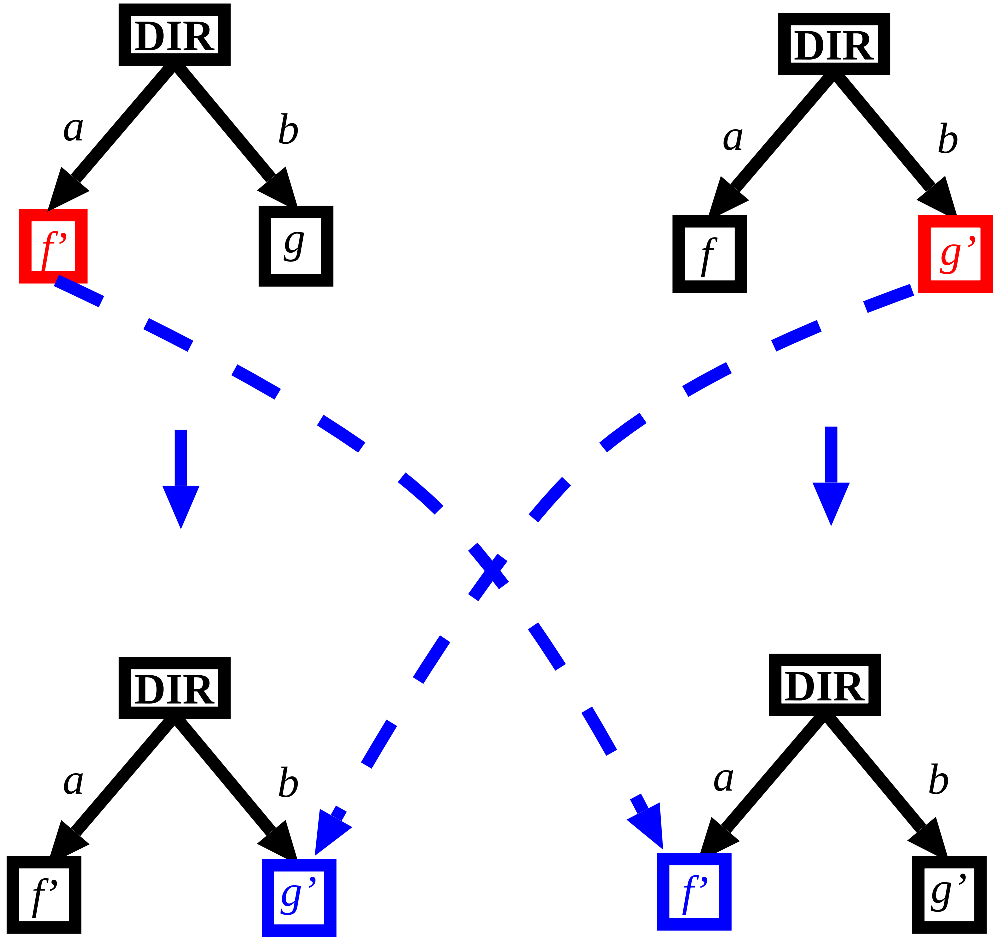
1. Develop a solid conceptual foundation for the task of file synchronization.
2. Build a robust, portable, cross-platform synchronization tool.

Demo

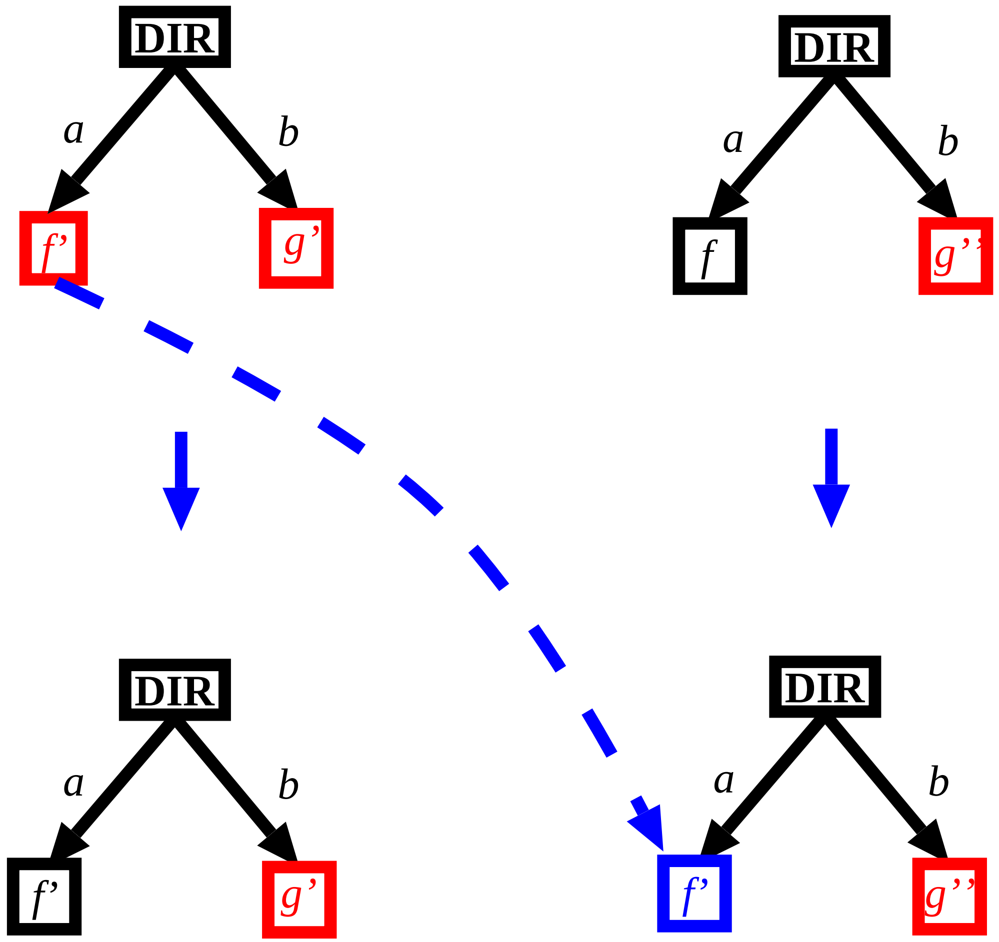
Foundations

Synchronization

A synchronizer should propagate changes...

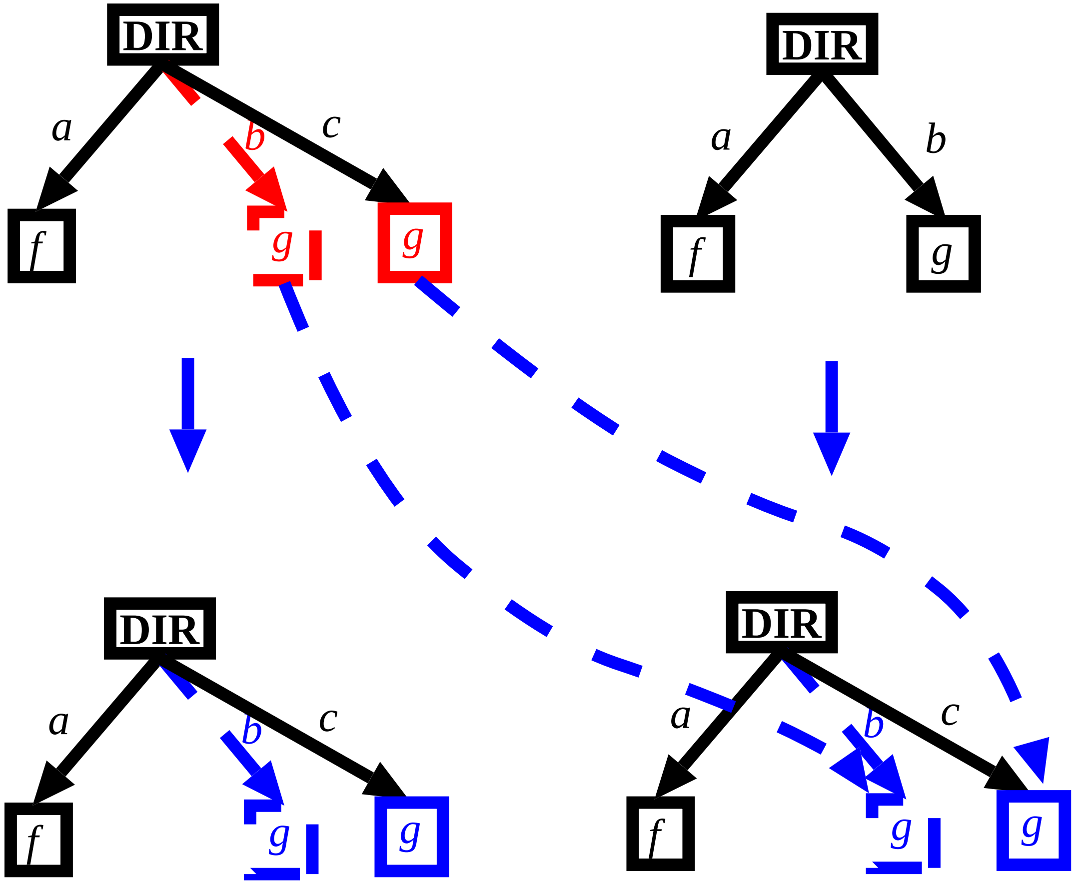


... as long as they do not conflict:

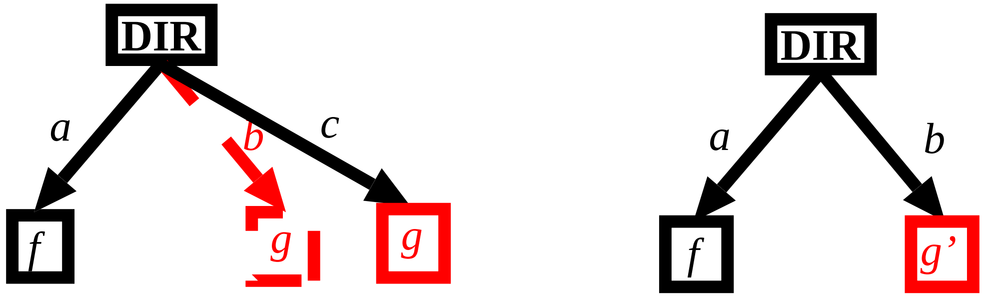


A less obvious example

Many synchronizers treat renaming a file as a deletion plus a creation:

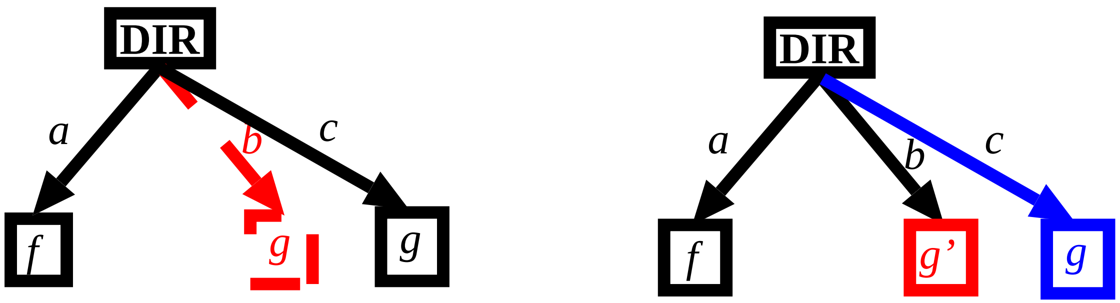


But if a file gets renamed on one side and modified on the other, what should the synchronizer do?

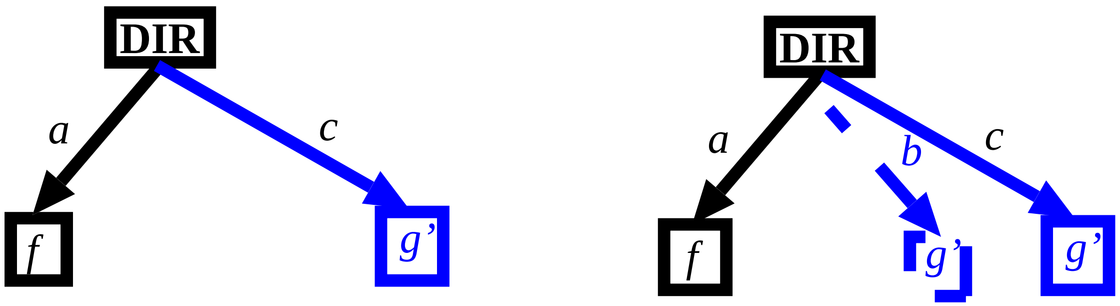


Three reasonable possibilities:

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



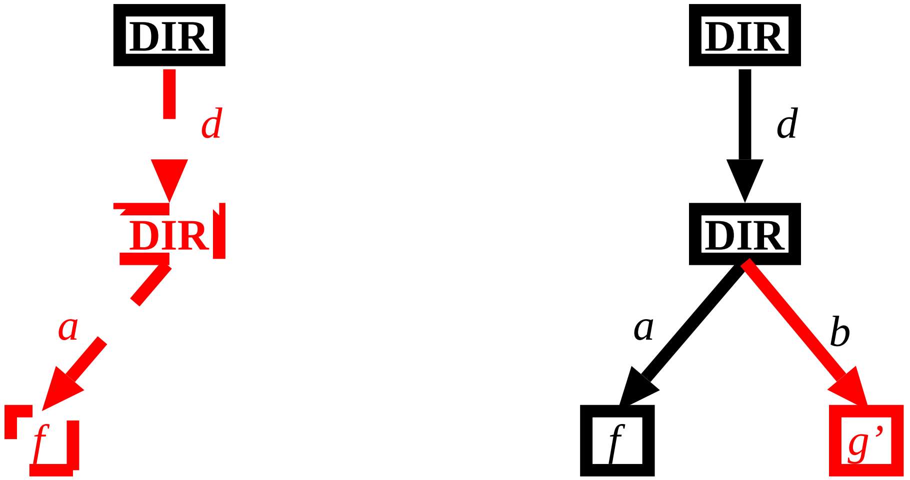
2. Modify the file in the first replica and move it in the second



3. 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 ($\mathrm{d.a}$) should be deleted from the second replica, leaving just the file ($\mathrm{d.b}$) and its parent directory (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

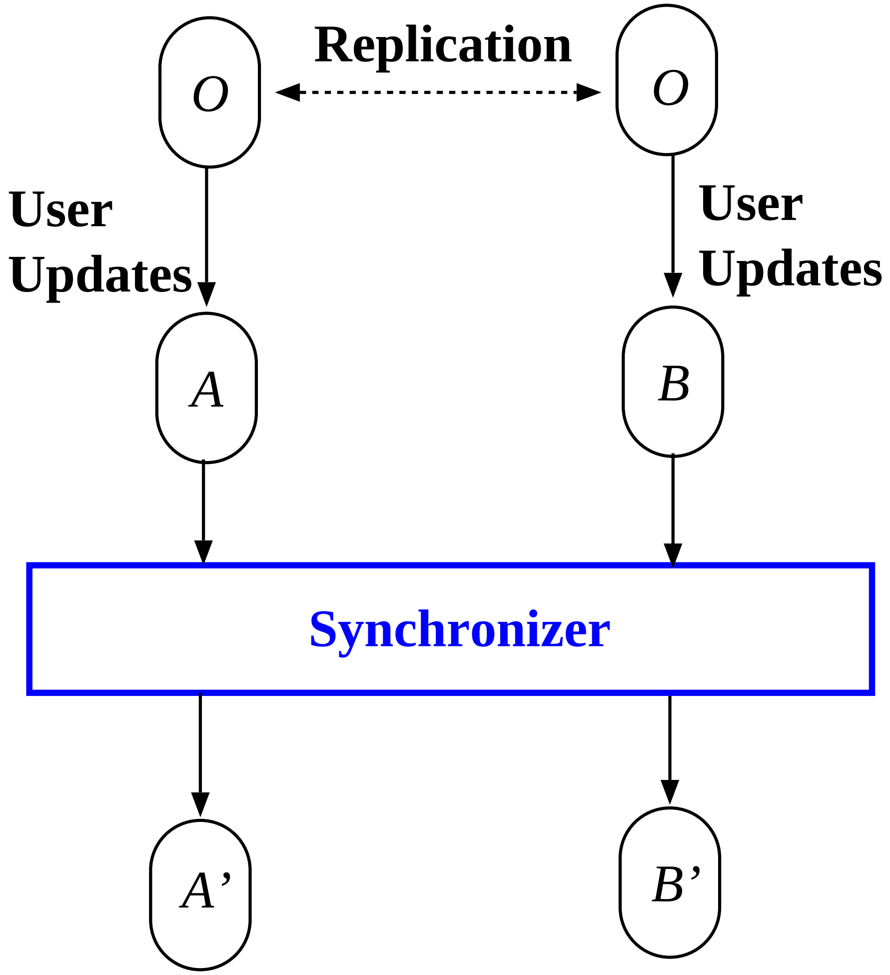
What we want...

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

Organizing principles

- 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 ($\mathrm{d}.\mathrm{e}.\mathrm{f}$) and for concatenating paths ($\mathrm{p}.\mathrm{q}$).
- A path q is called an ancestor of p, written $\mathrm{q} \leq \mathrm{p}$ if $\mathrm{p} = \mathrm{q}.\mathrm{r}$ for some (possibly empty) path r.
- A file is a value drawn from some uninterpreted set $\mathcal{F}$ (e.g., bytestrings).
- 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:

$$
\begin{aligned}
\mathcal{FS} \quad = \quad & \{\, \mathbf{S} \in \mathcal{P} \to (\mathcal{F} \cup \{\mathsf{DIR}, \perp\}) \ \mid \\
& \quad \forall \mathbf{p}, \mathbf{x}.\ \ \mathbf{S}(\mathbf{p}.\mathbf{x}) \neq \perp \ \Rightarrow\ \mathbf{S}(\mathbf{p}) = \mathsf{DIR} \,\}
\end{aligned}
$$

- Write $\mathbf{S}/\mathbf{p}$ ("$\mathbf{S}$ after $\mathbf{p}$") for $\lambda \mathbf{q}.\ \mathbf{S}(\mathbf{p}.\mathbf{q})$.

Conflicts

A path p is called a conflict path if

1. $A(p) \neq O(p)$
   $B(p) \neq O(p)$
   $A(p) \neq B(p)$

Conflicts

A path p is called a conflict path if some $q \le p$ is a conflict path or

1. $A(p) \neq O(p)$
   $B(p) \neq O(p)$
   $A(p) \neq B(p)$

Conflicts

A path $\mathbf{p}$ is called a conflict path if

1. $\mathbf{A}/\mathbf{p} \neq \mathbf{O}/\mathbf{p}$
   $\mathbf{B}/\mathbf{p} \neq \mathbf{O}/\mathbf{p}$
   $\mathbf{A}/\mathbf{p} \neq \mathbf{B}/\mathbf{p}$
2. $\mathbf{A}(\mathbf{p}) \neq$ DIR or $\mathbf{B}(\mathbf{p}) \neq$ DIR

Conflicts

A path p is called a conflict path if some $q \leq p$ is a conflict path or

1. $A/p \neq O/p$
   $B/p \neq O/p$
   $A/p \neq B/p$

2. $A(p) \neq$ DIR or $B(p) \neq$ DIR

First Specification

A synchronizer implementation is a relation

$$\text{snc} \in \underbrace{\mathcal{FS} \times \mathcal{FS} \times \mathcal{FS}}_{\text{inputs}} \times \underbrace{\mathcal{FS} \times \mathcal{FS}}_{\text{outputs}}$$

It is said to be correct if, for all $\mathbf{O}$, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{A}'$, and $\mathbf{B}'$ with $\text{snc}(\mathbf{O}, \mathbf{A}, \mathbf{B}, \mathbf{A}', \mathbf{B}')$, and for all paths $\mathbf{p}$,

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

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

(i.e., "if the user makes a change, it will not be overwritten")

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

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

(i.e., "the synchronizer only propagates real changes")

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

Observations

Note that 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. (We'll come back to this.)

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.

Partial 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?

We need to maintain a (fictitious) filesystem recording the "last synchronized state" of each path:

$$\mathsf{LastSync}(O, A', B') = \lambda p.\ \begin{cases} A'(p) & \text{if } A'(p) = B'(p) \\ O(p) & \text{otherwise} \end{cases}$$

Strategies for update detection

- Exact update detector

  $$\text{dirty}(p) \quad \text{iff} \quad \langle\text{current state of } p\rangle \neq O(p)$$

- Modtime update detector (for Unix)

  $$\begin{aligned} \text{dirty}(p) \quad \text{iff} \quad & \text{for some ancestor } q \text{ of } p, \\ & \text{modtime}(q) > \langle\text{last sync time}\rangle \end{aligned}$$

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

- Modtime-inode update detector (for Unix)

  $$\begin{aligned} \text{dirty}(p) \quad \text{iff} \quad & \text{modtime}(p) > \langle\text{last sync time}\rangle \\ \text{or} \quad & \text{inode}(p) \neq \text{inode}_O(p) \end{aligned}$$

- Trivial update detector

$$\mathsf{dirty}(p) = \mathbf{true}$$

Our present specification is committed to the exact detector, in that it requires that the original state of the filesystem be an input to the synchronizer.

In practice, exact update detection is rather expensive—it is better to use some conservative (safe) approximation that can be checked faster.

To reflect this refinement in the specification, we need to

1. enrich our definition of filesystems with extra information such as modtimes, and
2. introduce a new type of “memories” to model the internal state of the synchronizer that is preserved from one run to the next.

Enriched Filesystems

A rich filesystem $\mathcal{S}$ is a pair of functions

$$\mathsf{content}_{\mathcal{S}} \in \mathcal{FS} \quad \text{and} \quad \mathsf{info}_{\mathcal{S}} \in \mathcal{I}$$

where $\mathcal{I}$ is the extra information that the filesystem maintains about files. (E.g., on Unix, $\mathcal{I}$ might be the set of functions from paths to modtimes and inode numbers.) We'll continue to write $\mathcal{S}(p)$ for $\mathsf{content}_{\mathcal{S}}(p)$.

Write $\mathcal{RFS}$ for the set of rich file systems.

Of course, any particular implementation will make use of this information in quite specific ways. But the specification doesn't need to be precise about $\mathcal{I}$: we only need to know that the OS maintains some invariant between $\mathsf{content}_{\mathcal{S}}$ and $\mathsf{info}_{\mathcal{S}}$.

This can be stated abstractly by a relation

$\text{follows} \subseteq \mathcal{RFS} \times \mathcal{RFS}$.

For example, the follows relation for Unix would capture the fact that, if S' follows S, then $S(p) \neq S'(p)$ implies $\text{info}_S(p) \neq \text{info}'_S(p)$.

Specification with Memories

A synchronizer implementation comprises

- a set $\mathcal{M}$ of "memories"
- an initial memory $M_{\text{init}} \in \mathcal{M}$
- a relation

$$\text{snc} \in \underbrace{\mathcal{M} \times \mathcal{RFS} \times \mathcal{RFS}}_{\text{inputs}} \times \underbrace{\mathcal{M} \times \mathcal{RFS} \times \mathcal{RFS}}_{\text{outputs}}$$

It is said to be correct for rich filesystems A_{prev} and B_{prev}, memory M, and filesystem O if (informally)

for each A following A_{prev} and B following B_{prev},
if $\text{snc}(M, A, B, M' A', B')$,

- for all paths p, conditions (1)-(3) hold as before
- it is also correct for A', B', M', and $\text{LastSync}(O, A', B')$.

It is said to be correct if it is correct for

$$(\textsc{EmptyRfs},\ \textsc{EmptyRfs}, \textsc{EmptyFs}, \mathbf{M}_{\mathsf{init}})$$

More Precisely...

Formally, correctness is a coinductively defined predicate:

The correct set of a synchronizer implementation is the largest set

$$\mathsf{CO} \subseteq \mathcal{RFS} \times \mathcal{RFS} \times \mathcal{M} \times \mathcal{FS}$$

such that

for each $(A_{\mathsf{prev}}, B_{\mathsf{prev}}, M, O) \in \mathsf{CO}$
for each A following A_{prev} and B following B_{prev},
if $\mathsf{snc}(M, A, B, M' A', B')$,

- for all paths p, conditions (1)-(3) hold as before
- $(A', B', M', \mathsf{LastSync}(O, A', B')) \in \mathsf{CO}$.

A synchronizer implementation is correct if its correct set contains

$$(\textsc{EmptyRFs}, \textsc{EmptyRFs}, M_{\mathsf{init}}, \textsc{EmptyFs}).$$

Observations

We observed earlier that the simple specification did not force the synchronizer to propagate any changes at all.

Similarly, the new specification does not force the synchronizer to remember anything at all.

Of course, the same non-functional requirement as before remains in place: a “good” synchronizer implementation should remember information that helps it propagate as many changes as it can.

Extensions and variations

The simple specification we've seen can be enriched in numerous ways, e.g.

- Data synchronization
- Multiple replicas
- Heuristic or interactive merging of overlapping updates
- Permission bits, symbolic links, etc.

Data synchronizers

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

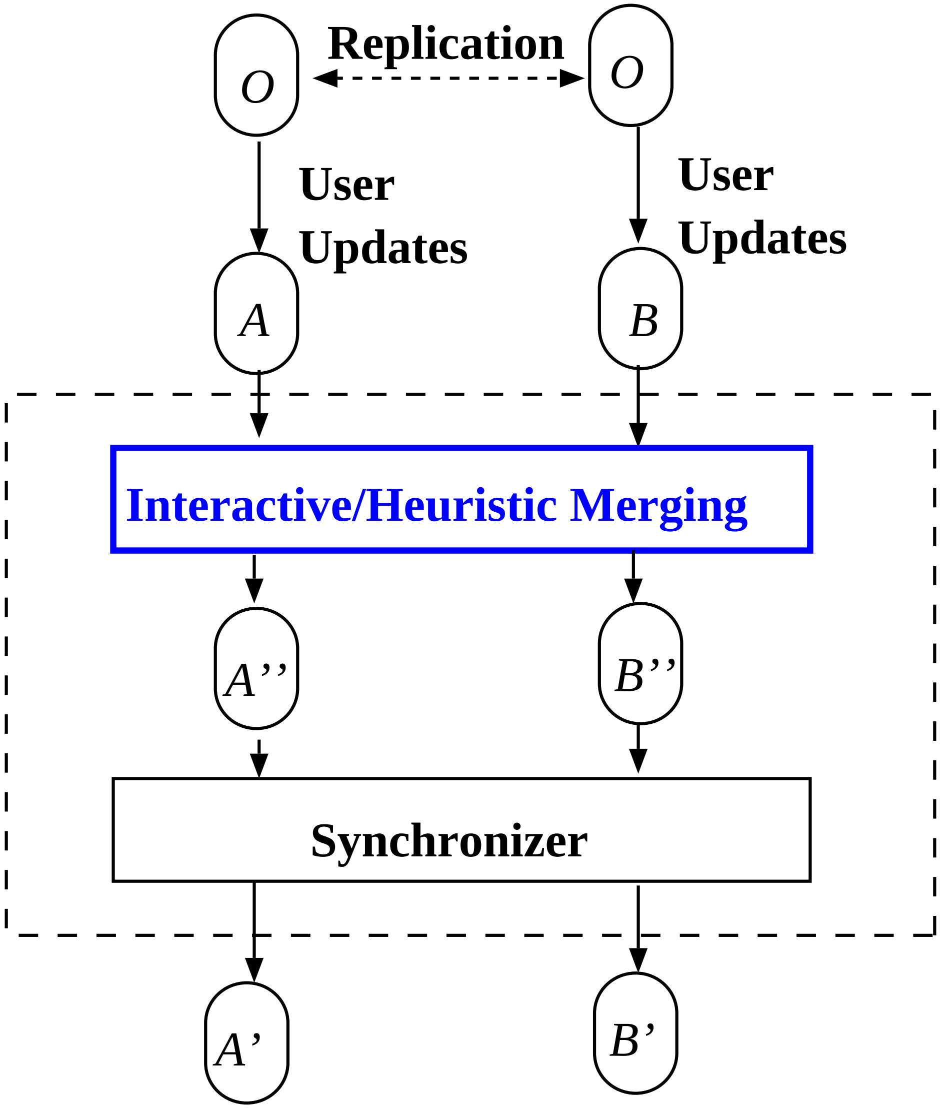
Multiple replicas

For notational simplicity, we've treated just the two-replica case. Everything extends straightforwardly to n replicas if we require that all replicas participate in every synchronization.

If we want to sometimes synchronize just a subset of the replicas, then the problem becomes trickier.

$\longrightarrow$ version vectors?

Incorporating heuristic / interactive merging



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 symlink-string are considered part of the contents of the file as far as the synchronizer is concerned.

[Permission bits are handled similarly: just consider them as part of the "contents" of the file.]

Hard links (as well as more general graph structures) are more problematic.

Examples

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

- Microsoft Briefcase (Windows)...

  Yes, modulo several bugs and some 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

Implementation

Unison

Our own synchronizer, called Unison, aims for robustness and portability.

- Runs on Windows and several flavors of Unix (and allows cross- platform synchronization between Windows and Unix filesystems)
- In daily use by 300+ people, synchronizing up to ~1Gb filesystems
- Deals with symlinks, file permissions, etc., etc.
- Uses modtime-inode update detection strategy (on Unix) for good performance with few “false positives”
- Tuned for high-bandwidth (ethernet) and medium-bandwidth (PPP) connections
- Tunnels over ssh for security (can also use rsh or raw sockets)

- Easy install (single executable file, no administrative privileges required)
- Source code available under GPL ($\sim$15K lines of O'Caml)
- Design strongly influenced by the specification discussed earlier (and vice versa)

Robustness issues

- Atomic changes to filesystems
- Resilience to concurrent filesystem modifications
- Two-phase locking of archive files
- etc, etc.

Heterogeneity

- Synchronizing permission bits
- (and a million and one other annoying little issues, e.g., case-insensitive filenames, ...)

Want to play?...

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