Unison

A file synchronizer and its specification

TACS 2001, Sendai

Benjamin C. Pierce
University of Pennsylvania

joint work with
Trevor Jim (AT&T)
Jérôme Vouillon (CNRS)

"First, do no harm..." —Hippocrates

Synchronization is all around us...

increasing distribution $\Longrightarrow$ replicated data

increasing mobility $\Longrightarrow$ disconnected updates

$\Downarrow \quad \Downarrow \quad \Downarrow$

synchronization

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.

Personnel:

Benjamin C. Pierce (Penn)
Trevor Jim (Penn $\longrightarrow$ AT&T)
Jérôme Vouillon (Penn $\longrightarrow$ CNRS),

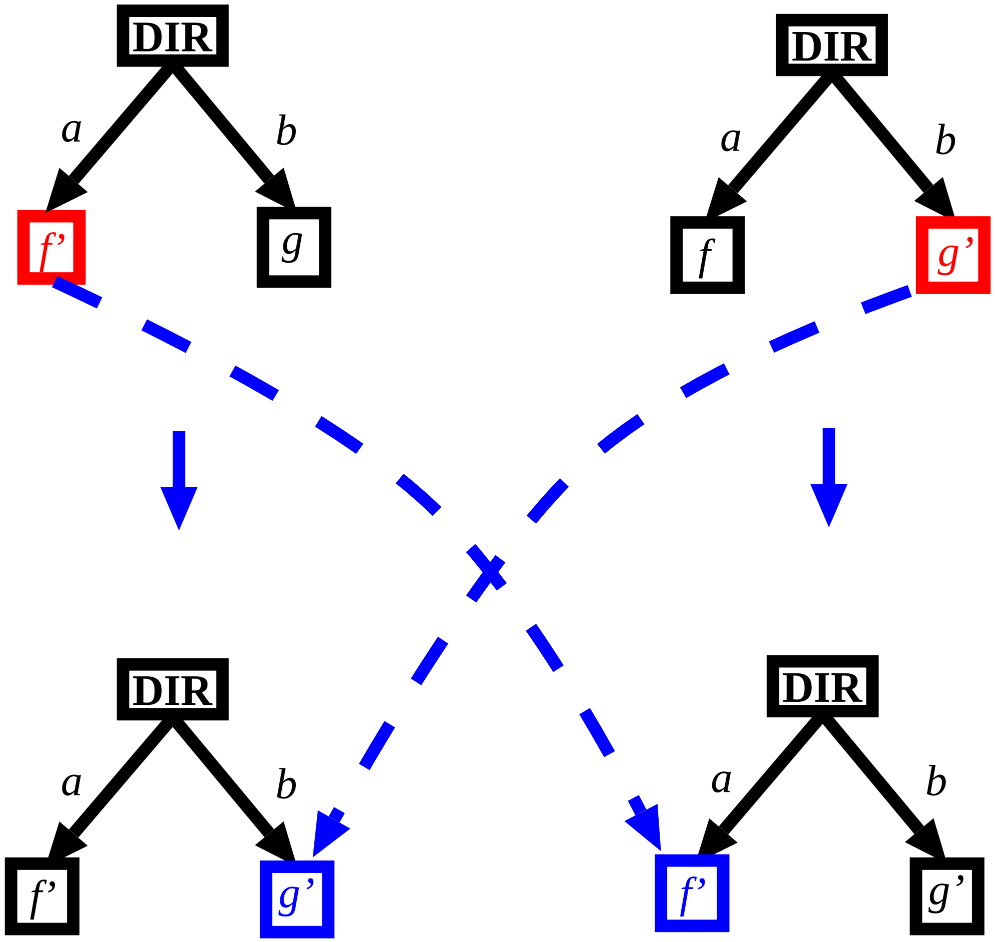
with
Sundar Balasubramaniam (Jareva Tech.), Mattheiu Goulay, Sylvain Gommier, Sylvain Roy (Ecole Polytechnique), Zhe Yang (Penn). Jacques Garrigue (Kyoto), etc.

Demo

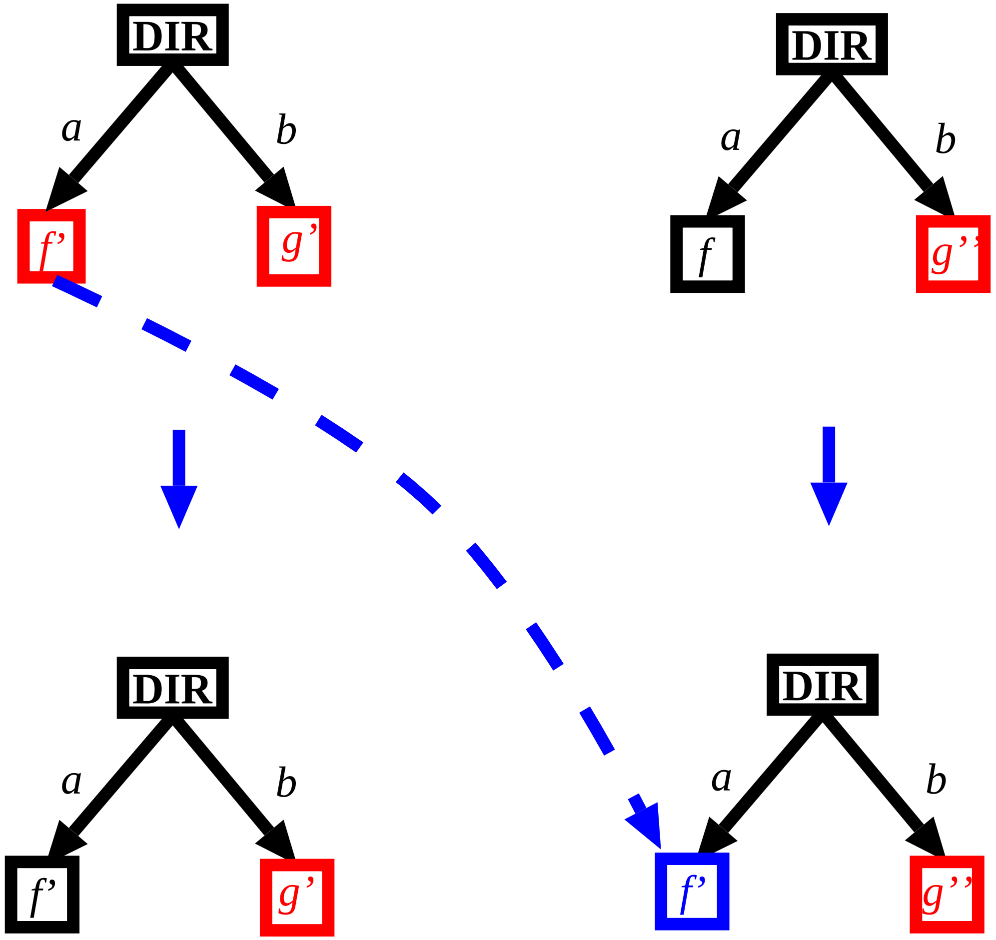
Foundations

Synchronization (a simple example)

A synchronizer should propagate changes...

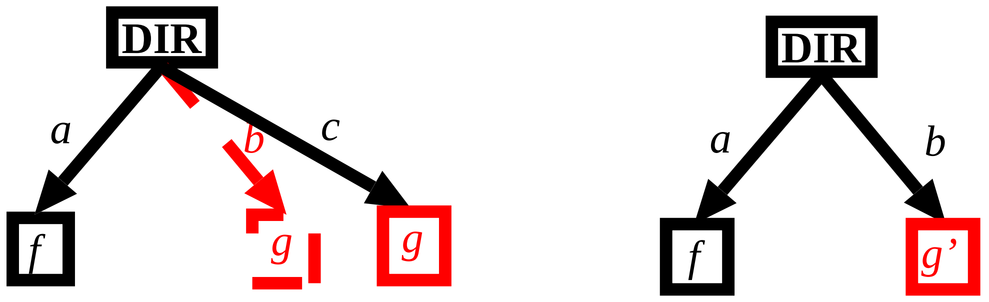


... as long as they do not conflict:



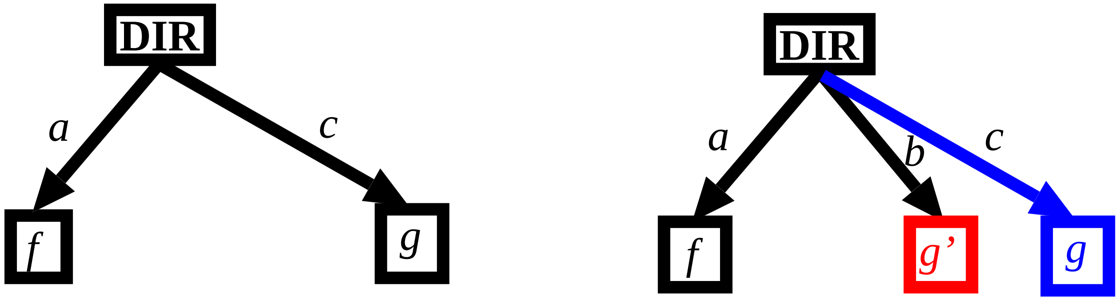
A more interesting example

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

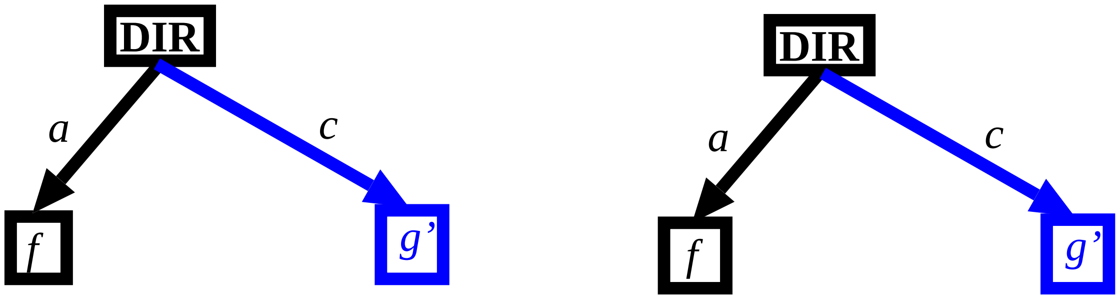


Three reasonable possibilities:

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



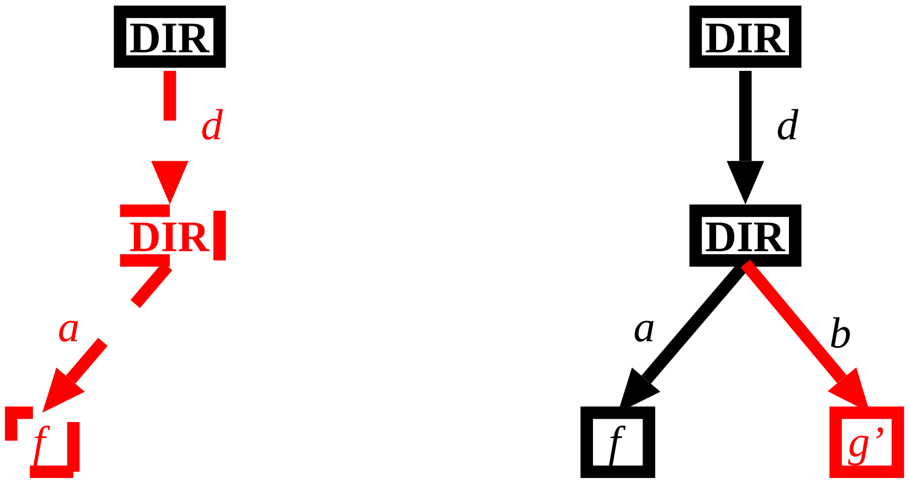
2. Modify the file in the first replica and move it in the second



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 ($\mathrm{d.a}$) 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

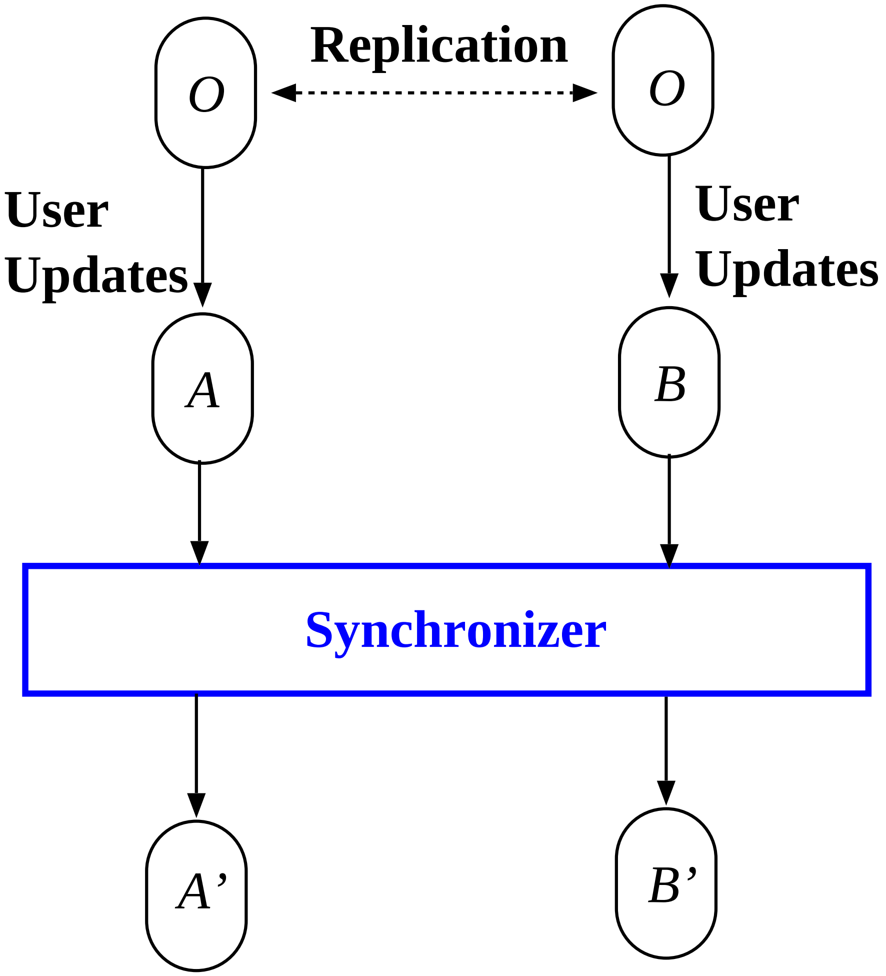
What we want...

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

Organizing principles

- Start by trying to specify just one synchronizer (Unison) 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

- Assume
  - a set $\mathcal{L}$ of labels

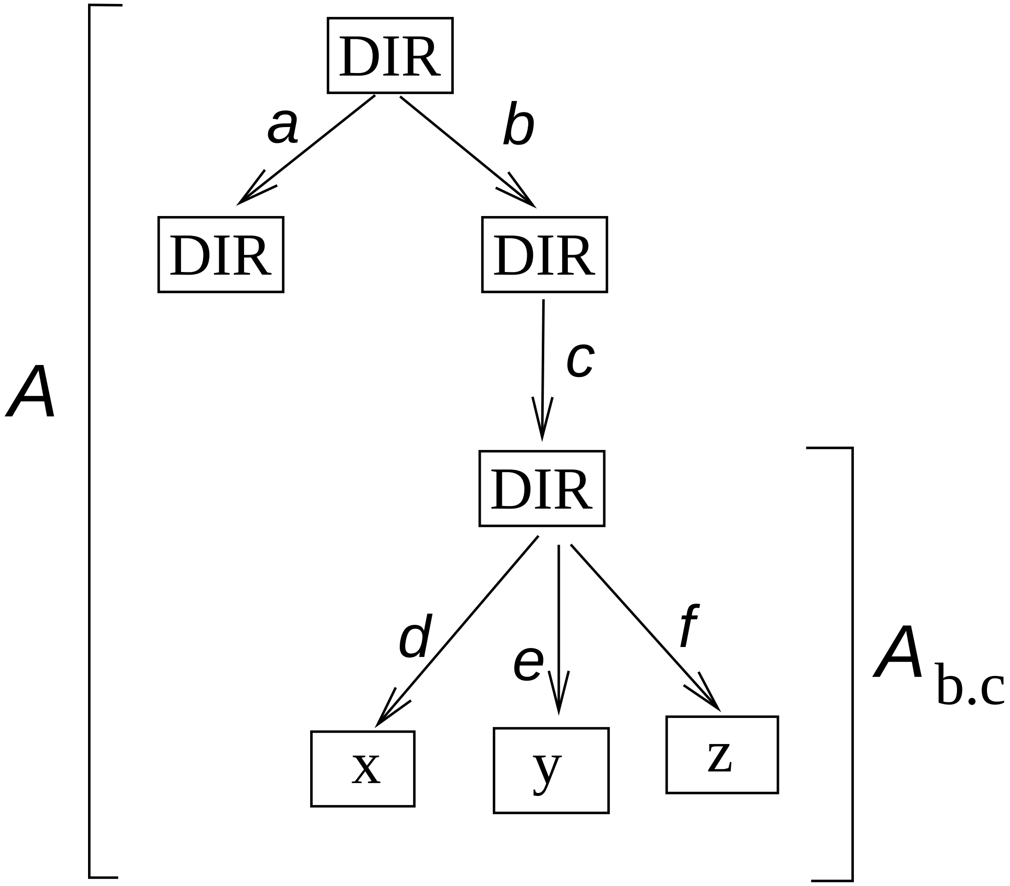
    e.g. $\mathcal{L} = \{\text{FILE}(c) \mid c \in \text{byte strings}\} \cup \{\text{DIR}\}$
  - a set $\mathcal{N}$ of names

    e.g. $\mathcal{N} = \{\text{strings not containing /}\}$
- A path is a (possibly empty) sequence of names.
- An abstract filesystem is an unordered tree with nodes labels drawn from $\mathcal{L}$ and edge labels drawn from $\mathcal{N}$.

  Formally, the set $\mathcal{FS}$ of abstract filesystems is the smallest set such that

  1. $\bot \in \mathcal{FS}$
  2. if $l \in \mathcal{L}$ and $ch \in \mathcal{N} \to \mathcal{FS}$, then $\text{NODE}(c, ch) \in \mathcal{FS}$

Write $\mathbf{A_p}$ ("$\mathbf{A}$ after $\mathbf{p}$") for the sub-filesystem of $\mathbf{A}$ rooted at path $\mathbf{p}$



Comparing Filesystems

The specification uses both "shallow" and "deep" comparisons between filesystems.

- Filesystems F and G have the same root label, written $F \sim G$, if either $F = G = \perp$ or $F = \mathsf{NODE}(l, ch)$ and $G = \mathsf{NODE}(l, ch')$ for the same l.
- F and G are equal, written $F = G$, if they are exactly the same tree.

Conflicts

Key question: What is a conflict?

Conflicts

Key question: What is a conflict?

Our answer: A conflict occurs when the two replicas have different labels at some path, and both have been changed.

Conflicts

Key question: What is a conflict?

Our answer: A conflict occurs when the two replicas have different labels at some path, and both have been changed.

Formally, we say that filesystems $\mathbf{O}$, $\mathbf{A}$, and $\mathbf{B}$ conflict at path $\mathbf{p}$ if

	$\neg(\mathbf{A}_\mathbf{p} \sim \mathbf{B}_\mathbf{p})$	"$\mathbf{A}$ and $\mathbf{B}$ are different at $\mathbf{p}$"
and	$\neg(\mathbf{A}_\mathbf{p} = \mathbf{O}_\mathbf{p})$	"$\mathbf{A}$ has been changed at or below $\mathbf{p}$"
and	$\neg(\mathbf{B}_\mathbf{p} = \mathbf{O}_\mathbf{p})$	"$\mathbf{B}$ 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 acceptable if, for all paths p:

(1) if $\neg(A_p \sim O_p)$, then $A'_p \sim A_p$
if $\neg(B_p \sim O_p)$, then $B'_p \sim B_p$
("don't overwrite user changes")

(2) if $\neg(A'_p \sim A_p)$, then $A'_p = B_p$
if $\neg(B'_p \sim B_p)$, then $B'_p = A_p$
("modify only by completely propagating user changes")

(3) if there is a conflict at p, then $A'_p = A_p$ and $B'_p = B_p$
("no changes at or below conflicting paths")

A synchronizer is safe if all its runs are acceptable.

Observation

Theorem (Laziness is safe): For every $\mathbf{O}$, $\mathbf{A}$, and $\mathbf{B}$, the run $(\mathbf{O}, \mathbf{A}, \mathbf{B}, \mathbf{A}, \mathbf{B})$ is safe.

Maximality

Of course, we ordinarily hope that a synchronizer will do something when it runs.

Call a run (O, A, B, A', B') maximal if it is safe and if A' and B' are "as similar as possible."

Maximality

Of course, we ordinarily hope that a synchronizer will do something when it runs.

Call a run (O, A, B, A', B') maximal if it is safe and if A' and B' are "as similar as possible."

Formally, for each path p, if there is any safe run (O, A, B, A'', B'') such that $A''_p \sim B''_p$, then $A'_p \sim B'_p$.

Now:

- Every run should be safe.
- Complete runs should also be maximal.

More Observations

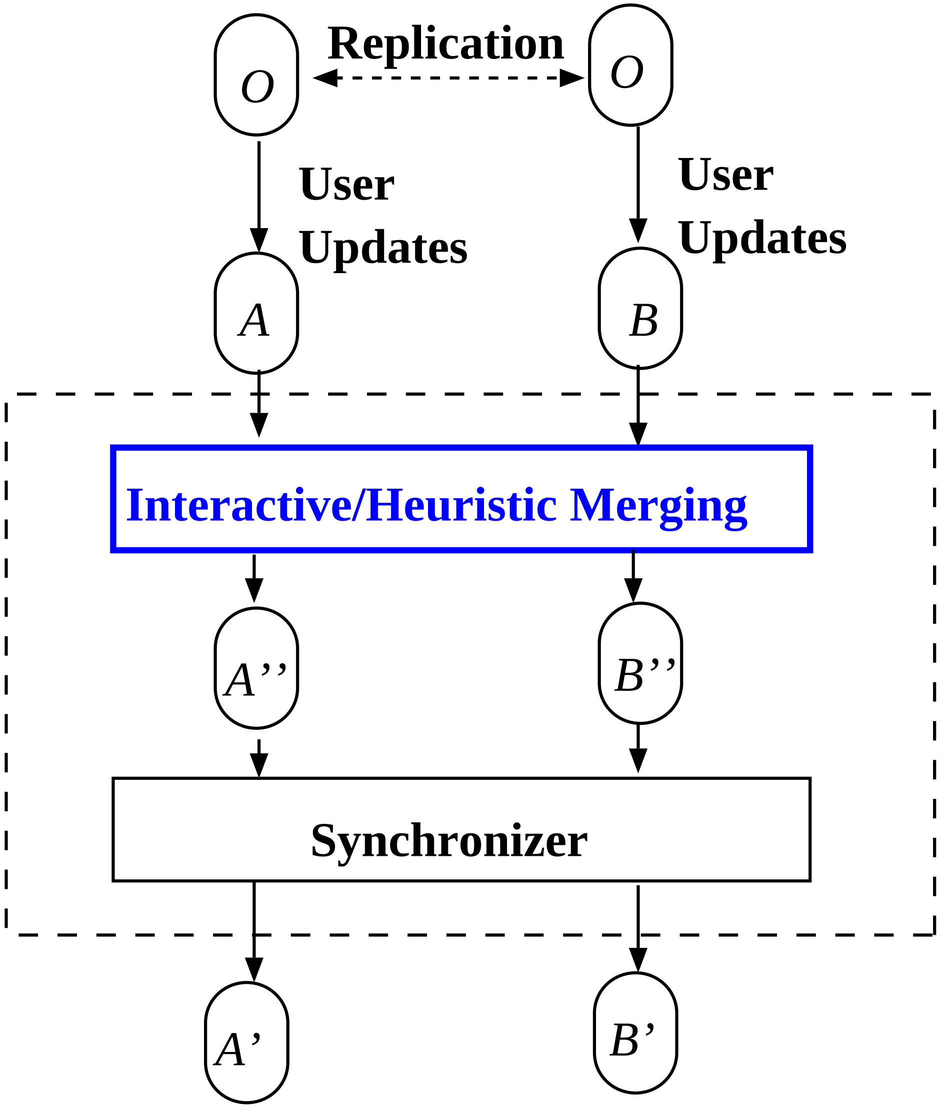
Theorem (Maximal runs are unique): For every $\mathbf{O}$, $\mathbf{A}$, and $\mathbf{B}$, there are unique $\mathbf{A}'$ and $\mathbf{B}'$ such that the run $(\mathbf{O}, \mathbf{A}, \mathbf{B}, \mathbf{A}', \mathbf{B}')$ is maximal.

Theorem (Synchronization degenerates to mirroring): For every $\mathbf{O}$ and $\mathbf{A}$, the run $(\mathbf{O}, \mathbf{A}, \mathbf{O}, \mathbf{A}, \mathbf{A})$ is maximal.

Theorem (Completeness in the absence of conflicts): If there are no conflicting paths in $\mathbf{O}$, $\mathbf{A}$, and $\mathbf{B}$, then there is a filesystem $\mathbf{C}$ such that $(\mathbf{O}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{C})$ is maximal.

Going Deeper

Incorporating heuristic / interactive merging



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” $\mathbf{O}$ 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" $\mathbf{O}$ on the next round of synchronization?

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

$$\mathbf{O}'_{\mathbf{p}} \sim \begin{cases} \mathbf{A}'_{\mathbf{p}} & \text{if } \mathbf{A}'_{\mathbf{p}} \sim \mathbf{B}'_{\mathbf{p}} \\ \mathbf{O}_{\mathbf{p}} & \text{otherwise} \end{cases}$$

Strictly speaking...

To deal with the possibility of machine failures during synchronization, we actually need to treat $\mathbf{O}'$ as an explicit output of the synchronizer (rather than calculating it externally, in the specification).

The specification is extended accordingly.

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, ACLs, etc.

View files as “pseudo directories” with various special children...

- contents (as before)
- permissions
- owner
- group
- etc.

Heterogeneity

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

This involves dealing with...

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

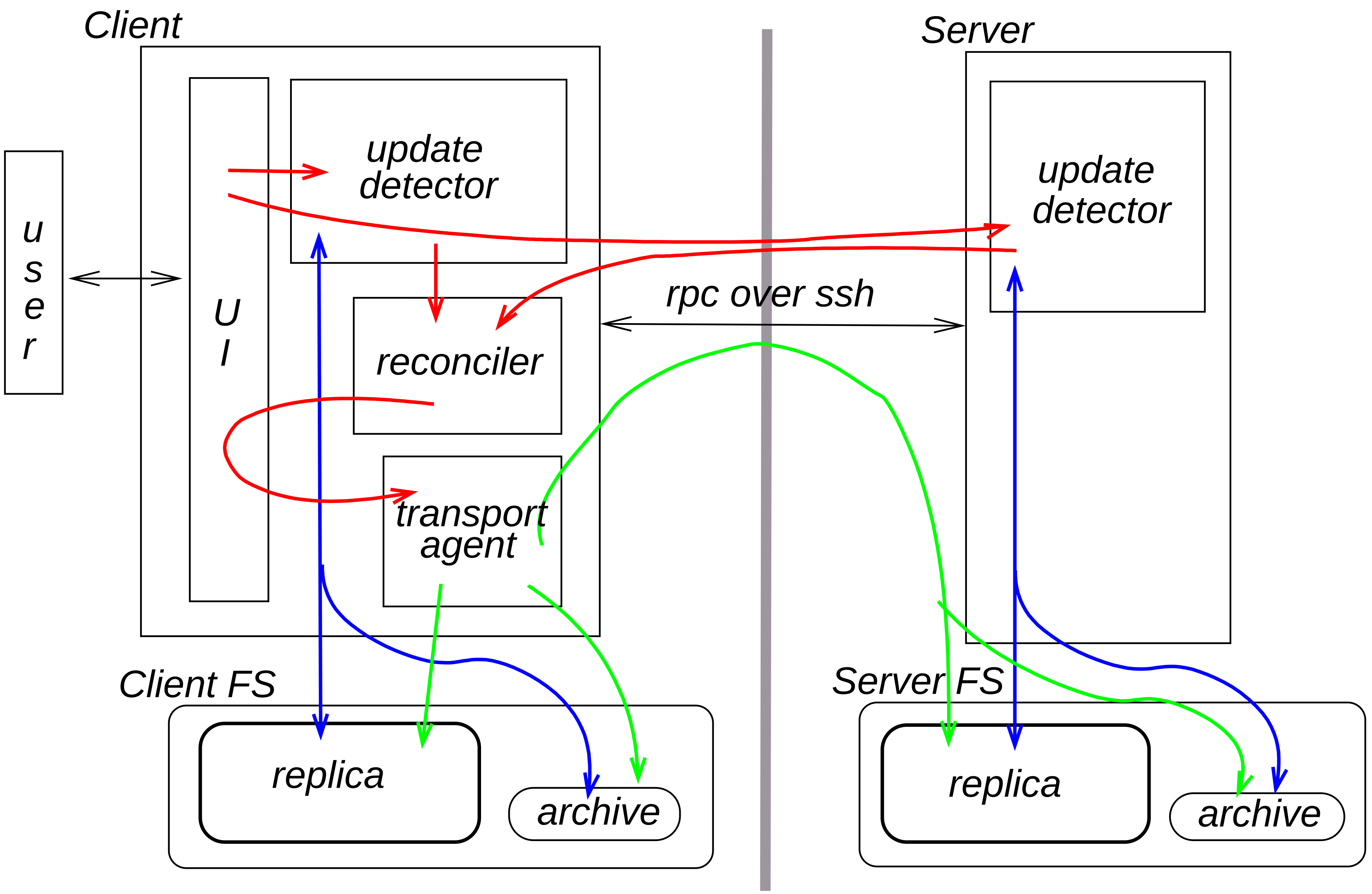
To achieve this, we need to change our goal to "synchronizing the common information" (and doing something reasonable with the rest).

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
- Incorporates an implementation of 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 (~500-3000 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 (naturally), plus a few unavoidable races

Formalization

We have formalized (in Coq) a significant part of the above development.

- Core definitions
- Specification (including several refinements not shown here)
- A "reference implementation"
  - Purely functional
  - Manipulating simple tree structures rather than concrete filesystems
  - Ignoring distribution, user interface, etc., etc.

  But...
  - Nontrivial (few hundred lines)
  - Control structure recognizably similar to actual Unison implementation
  - Data structures very similar to actual implementation
- Proof that reference implementation meets specification

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

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 tricky 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.

A Challenge!

As a simple way of getting started with synchronizing ordered structures, suppose we want to synchronize lists. (For the sake of concreteness, suppose that the lists represent to-do lists.)

A list synchronizer is a program that takes three lists, $\mathbf{O}$, $\mathbf{A}$, and $\mathbf{B}$, and yields lists $\mathbf{A}'$ and $\mathbf{B}'$.

The challenge:

- Give a precise characterization of which synchronizer runs $(\mathbf{O}, \mathbf{A}, \mathbf{B}, \mathbf{A}', \mathbf{B}')$ are acceptable.
- Argue that your definition agrees with people's natural expectations of how a list synchronizer should behave.

Arguing for “pragmatic reasonableness”...

How does one argue that some specification “agrees with people’s natural expectations”?

1. By checking how it works on lots of examples.

Arguing for "pragmatic reasonableness"...

How does one argue that some specification "agrees with people's natural expectations"?

1. By checking how it works on lots of examples.

2. By stating general properties that a list synchronizer should have, arguing that they are reasonable, and then showing that they are implied by the specification.

Arguing for "pragmatic reasonableness"...

How does one argue that some specification "agrees with people's natural expectations"?

1. By checking how it works on lots of examples.
2. By stating general properties that a list synchronizer should have, arguing that they are reasonable, and then showing that they are implied by the specification.

   E.g...

   - Laziness is good: $(\mathbf{O}, \mathbf{A}, \mathbf{B}, \mathbf{A}, \mathbf{B})$ is acceptable.
   - Synchronization degenerates to mirroring: $(\mathbf{O}, \mathbf{A}, \mathbf{O}, \mathbf{A}, \mathbf{A})$ is acceptable.
   - List sync coincides with set sync when ordering is ignored...

What is set synchronization?

Proposal...

Let $\mathbf{O}$, $\mathbf{A}$, and $\mathbf{B}$ be sets. Then $(\mathbf{O}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{C})$ is acceptable iff

$$\mathbf{C} = (\mathbf{O} \setminus \mathsf{deletedFromA} \setminus \mathsf{deletedFromB}) \cup \mathsf{addedToA} \cup \mathsf{addedToB}$$

where

$$\begin{aligned}
\mathsf{deletedFromA} &= \mathbf{O} \setminus \mathbf{A} \\
\mathsf{deletedFromB} &= \mathbf{O} \setminus \mathbf{B} \\
\mathsf{addedToA} &= \mathbf{A} \setminus \mathbf{O} \\
\mathsf{addedToB} &= \mathbf{B} \setminus \mathbf{O}.
\end{aligned}$$

Now, we may choose to demand that list synchronization should coincide with set synchronization, in the sense that if $(\mathbf{O}, \mathbf{A}, \mathbf{B}, \mathbf{A}', \mathbf{B}')$ is an acceptable run of a list synchronizer, then $(\{\mathbf{O}\}, \{\mathbf{A}\}, \{\mathbf{B}\}, \{\mathbf{A}'\}, \{\mathbf{B}'\})$ is an acceptable run of a set synchronizer.

Is this a good idea?

You decide.

(Note, though, that this condition is pretty strong, since set synchronization can never encounter a conflict, while list synchronization certainly can.)

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

End of talk

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

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

- Modtime update detector (for Unix)

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

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

- Modtime-inode update detector (for Unix)

  $\text{dirty}(p)$ iff $\text{modtime}(p) > \langle\text{last sync time for } p\rangle$
  or $\text{inode}(p) \neq \text{inode}_{\mathcal{O}}(p)$