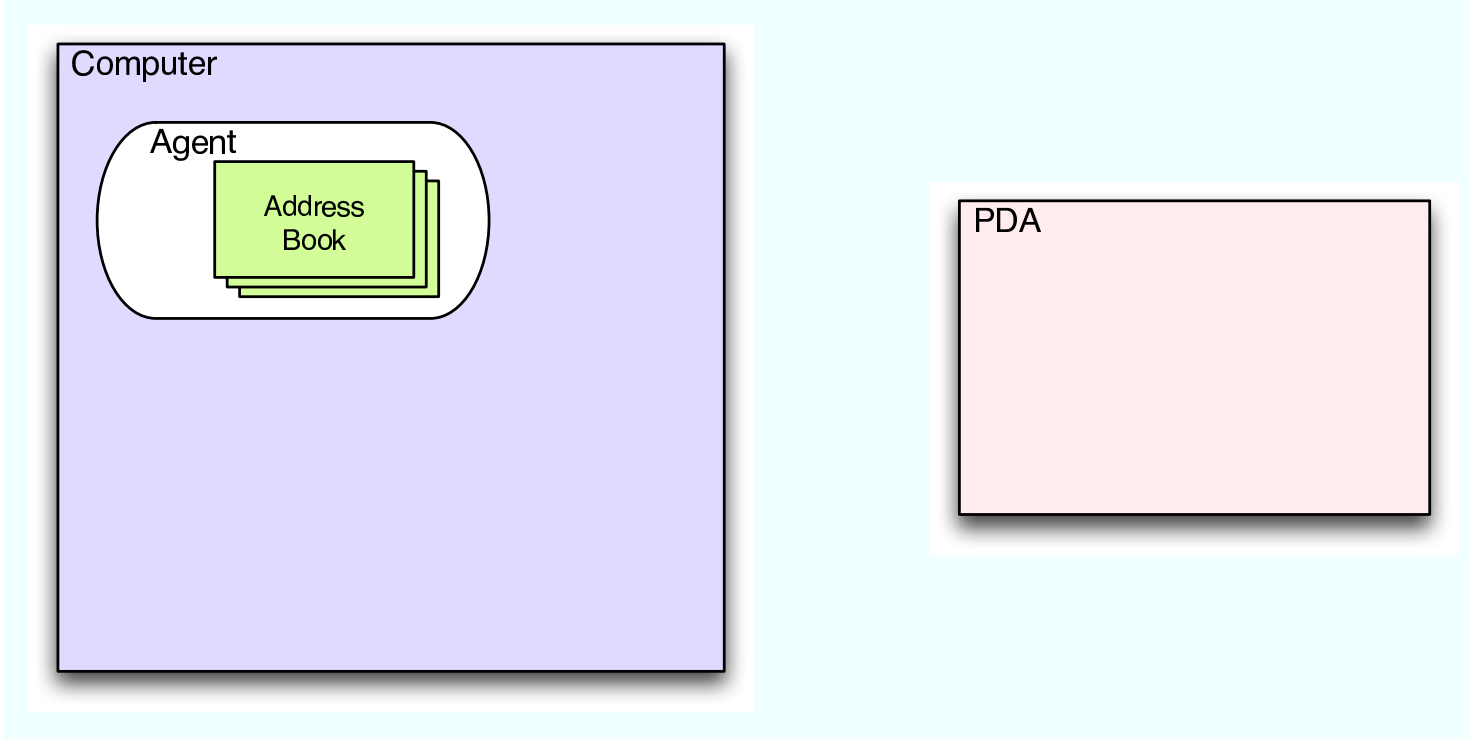
The Art of Reconciliation

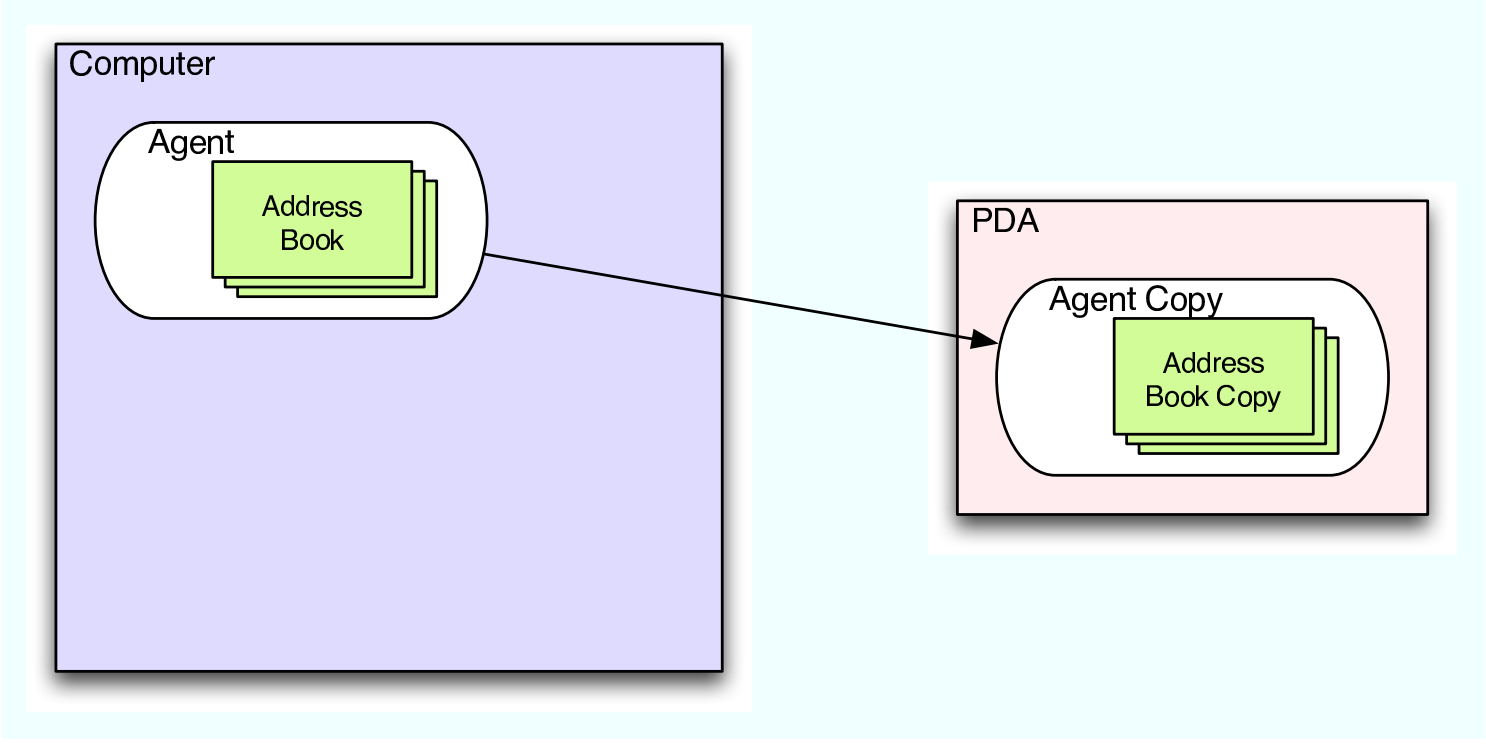
Benjamin C. Pierce

University of Pennsylvania

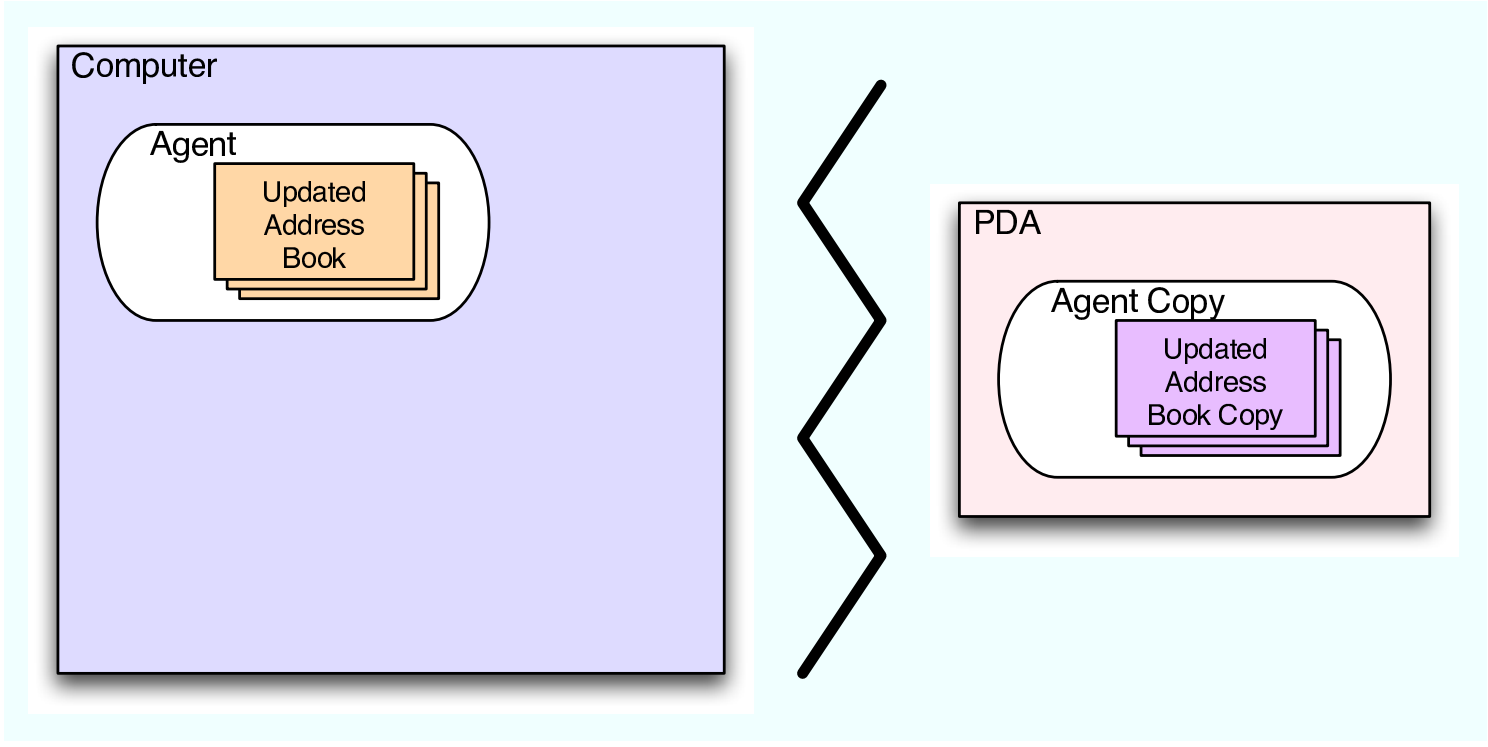
The Dangers of Replication



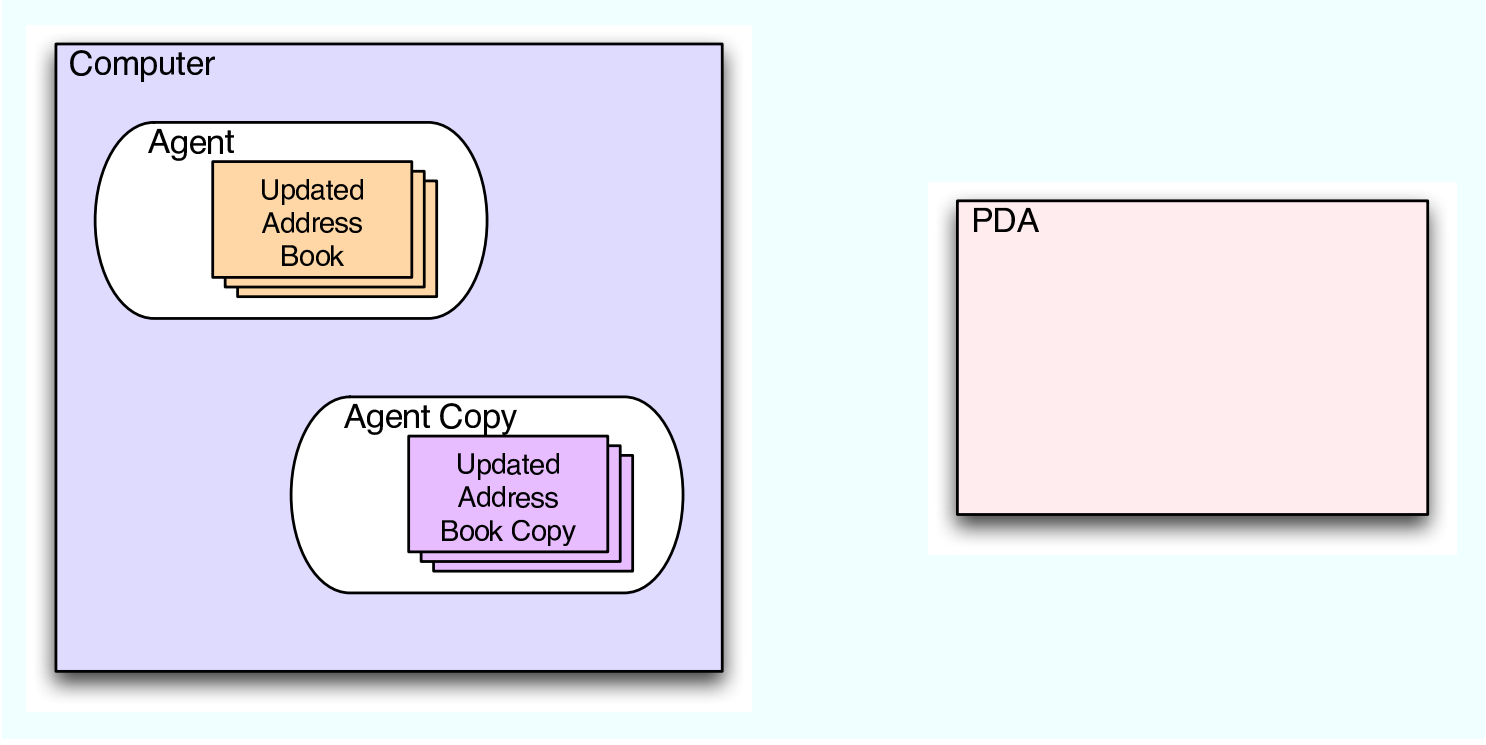
The Dangers of Replication



The Dangers of Replication



The Dangers of Replication



Optimistic Replication

- Many copies of data, held by geographically distributed hosts (mobile agents, databases, user filesystems, etc.)
- Any copy may be updated at any time
- Hosts occasionally reconcile (or synchronize) their states

Advantages of Optimism

- Availability: hosts can update their copies of data while disconnected
- Scalability: no "hot spots" for writes
- Quality control: updates can be "curated" before being allowed into a replica
- Visibility/atomicity control: a set of updates can be "kept local" until ready, then propagated to other hosts (cf. CVS)

Challenges of Optimism

Pragmatic issues:

- How to make sure updates get propagated in a timely manner (while dealing with failures, avoiding swamping the network or using too much local storage, etc.)?

  $\Longrightarrow$ Interesting when there are many replicas

Semantic issues:

- Precisely what does it mean to "synchronize" replicas?

  $\Longrightarrow$ Interesting when replicated data has nontrivial internal structure

Synchronizing States vs. Operations

Two basic approaches to semantics of synchronization:

- Operation-based synchronizers are given access to complete traces of all the operations performed on each replica of the data.
  Examples: Bayou, IceCube
- State-based synchronizers are given just the states of the replicas at particular moments in time.
  Examples: Unison, Harmony

Tradeoffs

Operation-based:

- Pro: temporal sequencing of operations on each replica visible to synchronizer
- Pro: operations can be chosen to encode high-level application semantics
- Con: require relatively tight coupling with applications
  $\Longrightarrow$ Appropriate for "synchronization middleware"

State-based:

- Con: less information available at sync time
- Pro: applications need not be "synchronization aware"
  $\Longrightarrow$ Appropriate for loosely coupled architectures

The Harmony Project

Research goal:

Build a generic synchronization framework for structured data stored as XML documents

The Harmony Project

Research goal:

Build a generic synchronization framework for structured data stored as XML documents

For this talk:

Focus on semantic issues

... particular, on Harmony's schema-directed synchronization algorithm

Running Example

An XML Address Book

```
<xcard>
  <vcard>
    <n>Rocco</n>
    <org>Edinburgh</org>
    <email>denicola@dcs.ed</email>
  </vcard>
  <vcard>
    <n>Davide</n>
    <org>Edinburgh</org>
    <email>sad@dcs.ed</email>
  </vcard>
</xcard>
```

An Updated Address Book

```
<xcard>
  <vcard>
    <n>Rocco</n>
    <org>Firenze</org>
    <email>denicola@dsi.unifi</email>
  </vcard>
  <vcard>
    <n>Davide</n>
    <org>Edinburgh</org>
    <email>sad@dcs.ed</email>
  </vcard>
</xcard>
```

Another Update

```
<xcard><vcard>
          <n>Rocco</n>
          <org>Edinburgh</org>
          <email>denicola@dcs.ed</email>
        </vcard>
        <vcard>
          <n>Davide</n>
          <org>Bologna</org>
          <email>Davide@cs.unibo</email>
        </vcard>
</xcard>
```

[Note that the formatting of the XML also changed a little in this version.]

Synchronization

How do we reconcile these changes to get back to a single, unified address book?

Synchronization: First Try

Using a textual merging tool like `diff3` doesn't get us very far because of the "insignificant" formatting changes in the second variant...

Synchronization: First Try

```
diff3 -m egA egO egB
```

```
<<<<<<< egA
<xcard>
  <vcard>
    <n>Rocco</n>
    <org>Firenze</org>
    <email>denicola@dsi.unifi</email>
  </vcard>
  <vcard>
    <n>Davide</n>
    <org>Edinburgh</org>
    <email>sad@dcs.ed</email>
  </vcard>
||||||| egO
<xcard>
  <vcard>
    <n>Rocco</n>
    <org>Edinburgh</org>
    <email>denicola@dcs.ed</email>
  </vcard>
  <vcard>
    <n>Davide</n>
    <org>Edinburgh</org>
    <email>sad@dcs.ed</email>
  </vcard>
=======
<xcard><vcard>
          <n>Rocco</n>
          <org>Edinburgh</org>
          <email>denicola@dcs.ed</email>
        </vcard>
        <vcard>
          <n>Davide</n>
          <org>Bologna</org>
          <email>Davide@cs.unibo</email>
        </vcard>
>>>>>>> egB
</xcard>
```

We Need A More Abstract View

The problem here is obvious: we want to synchronize our address books as trees, not as strings of characters.

Data Model

Trees

Harmony's core data model is the simplest we could think of — unordered, edge-labeled trees with all children of a node labeled differently.

(I.e., each tree is a partial function from labels to subtrees.)

$$\left\{\begin{array}{l}\texttt{name} \mapsto \{\texttt{Rocco} \mapsto \{\\ \texttt{org} \mapsto \{\texttt{Edinburgh} \mapsto \{\\ \texttt{email} \mapsto \{\texttt{denicola@dcs.ed} \mapsto \{\end{array}\right.$$

Lists

The list

$$[\mathtt{v}_1 \ldots \mathtt{v}_n]$$

is represented by the tree

$$\left\{\begin{array}{l} \texttt{*h} \mapsto \mathtt{v}_1 \\ \texttt{*t} \mapsto \left\{\begin{array}{l} \texttt{*h} \mapsto \mathtt{v}_2 \\ \texttt{*t} \mapsto \left\{\begin{array}{l} \ldots \mapsto \left\{\begin{array}{l} \texttt{*h} \mapsto \mathtt{v}_n \\ \texttt{*t} \mapsto \{ \end{array}\right. \end{array}\right. \end{array}\right. \end{array}\right.$$

XML

The XML element

```
<tag>
    subelt1 ... subeltn
</tag>
```

is represented by the tree

$$\left\{ \texttt{tag} \mapsto \left[\begin{array}{c} \langle \texttt{subelt1} \rangle \\ \vdots \\ \langle \texttt{subeltn} \rangle \end{array} \right. \right.$$

[XML with attributes is handled by a slightly more complex encoding.]

Back to the Example: Original State

$$
\left\{ \texttt{xcard} \mapsto \left[
\begin{array}{l}
\texttt{vcard} \mapsto \left[
\begin{array}{l}
\texttt{name} \mapsto \left[\texttt{Rocco} \mapsto [\right. \\
\texttt{org} \mapsto \left[\texttt{Edinburgh} \mapsto [\right. \\
\texttt{email} \mapsto \left[\texttt{denocola@dcs.ed} \mapsto [\right.
\end{array}
\right. \\
\texttt{vcard} \mapsto \left[
\begin{array}{l}
\texttt{name} \mapsto \left[\texttt{Davide} \mapsto [\right. \\
\texttt{org} \mapsto \left[\texttt{Edinburgh} \mapsto [\right. \\
\texttt{email} \mapsto \left[\texttt{sad@dcs.ed} \mapsto [\right.
\end{array}
\right.
\end{array}
\right.
\right.
$$

Back to the Example: First Variant

$$
\left\{ \texttt{xcard} \mapsto \left[
\begin{array}{ll}
\texttt{vcard} \mapsto & \left[
\begin{array}{l}
\texttt{name} \mapsto \left[\texttt{Rocco} \mapsto [\right. \\
\texttt{org} \mapsto \left[\texttt{Firenze} \mapsto [\right. \\
\texttt{email} \mapsto \left[\texttt{denocola@dsi.unifi} \mapsto [\right.
\end{array}
\right. \\
\texttt{vcard} \mapsto & \left[
\begin{array}{l}
\texttt{name} \mapsto \left[\texttt{Davide} \mapsto [\right. \\
\texttt{org} \mapsto \left[\texttt{Edinburgh} \mapsto [\right. \\
\texttt{email} \mapsto \left[\texttt{sad@dcs.ed} \mapsto [\right.
\end{array}
\right.
\end{array}
\right.
\right.
$$

Back to the Example: Second Variant

$$
\left\{ \texttt{xcard} \mapsto \left[
\begin{array}{ll}
\texttt{vcard} \mapsto & \left[
\begin{array}{ll}
\texttt{name} \mapsto & \left[\texttt{Rocco} \mapsto [\right] \\
\texttt{org} \mapsto & \left[\texttt{Edinburgh} \mapsto [\right] \\
\texttt{email} \mapsto & \left[\texttt{denocola@dcs.ed} \mapsto [\right]
\end{array}
\right] \\
\texttt{vcard} \mapsto & \left[
\begin{array}{ll}
\texttt{name} \mapsto & \left[\texttt{Davide} \mapsto [\right] \\
\texttt{org} \mapsto & \left[\texttt{Bologna} \mapsto [\right] \\
\texttt{email} \mapsto & \left[\texttt{Davide@cs.unibo} \mapsto [\right]
\end{array}
\right]
\end{array}
\right] \right\}
$$

Back to the Example: Merged State

$$
\left\{ \text{xcard} \mapsto \left[
\begin{array}{l}
\text{vcard} \mapsto \left[
\begin{array}{l}
\text{name} \mapsto \left[\text{Rocco} \mapsto [\right. \\
\text{org} \mapsto \left[\text{Firenze} \mapsto [\right. \\
\text{email} \mapsto \left[\text{denocola@dsi.unifi} \mapsto [\right.
\end{array}
\right. \\
\text{vcard} \mapsto \left[
\begin{array}{l}
\text{name} \mapsto \left[\text{Davide} \mapsto [\right. \\
\text{org} \mapsto \left[\text{Bologna} \mapsto [\right. \\
\text{email} \mapsto \left[\text{Davide@cs.unibo} \mapsto [\right.
\end{array}
\right.
\end{array}
\right.
\right.
$$

Alignment

Alignment

We solved the previous problem by noticing that what first appeared to be a conflict was actually a problem of alignment.

> alignment
> =
> deciding which parts of the replicas "correspond"

Alignment is the sine qua non of synchronization.

Alignment

We solved the previous problem by noticing that what first appeared to be a conflict was actually a problem of alignment.

alignment
=
deciding which parts of the replicas "correspond"

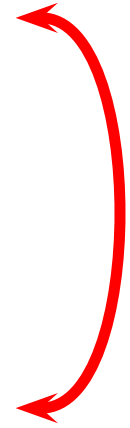
Alignment is the sine qua non of synchronization.

But we are not finished dealing with alignment yet...

Another Alignment Difficulty

Suppose that our second address book happens to get stored in reverse order...

```
<xcard><vcard>
          <n>Davide</n>
          <org>Bologna</org>
          <email>Davide@cs.unibo</email>
        </vcard>
        <vcard>
          <n>Rocco</n>
          <org>Edinburgh</org>
          <email>denicola@dcs.ed</email>
        </vcard>
</xcard>
```



Alignment Strategies

Two possibilities:

- Global: Compare the parts of the two replicas and try to come up with a "best alignment" of matching substructures
- Local: Reorganize both replicas so that the alignment becomes obvious.

Approaches to Alignment

Both have pros and cons:

- Global alignment:
  - Pro: Better at dealing with inherently "list-structured" data such as documents
  - Con: Heuristic
- Local alignment:
  - Pro: Simple
  - Con: Sometimes too simple

Past research has focused on global alignment strategies.

In Harmony, we have chosen to explore local strategies and see how far we can go.

Back to the Example

Before synchronization, we transform the concrete address book trees into "bushes" where each address record is reached by an edge labeled with its `name` component.

Transforming Away Ordering

$$
\left\{ \texttt{xcard} \mapsto \left[
\begin{array}{l}
\texttt{vcard} \mapsto \left[
\begin{array}{l}
\texttt{name} \mapsto \left[\texttt{Davide} \mapsto [\right. \\
\texttt{org} \mapsto \left[\texttt{Bologna} \mapsto [\right. \\
\texttt{email} \mapsto \left[\texttt{Davide@cs.unibo} \mapsto [\right.
\end{array}
\right. \\
\texttt{vcard} \mapsto \left[
\begin{array}{l}
\texttt{name} \mapsto \left[\texttt{Rocco} \mapsto [\right. \\
\texttt{org} \mapsto \left[\texttt{Edinburgh} \mapsto [\right. \\
\texttt{email} \mapsto \left[\texttt{denocola@dcs.ed} \mapsto [\right.
\end{array}
\right.
\end{array}
\right.
\right\}
$$

Transforming Away Ordering

$$\left\{ \texttt{xcard} \mapsto \left[\begin{array}{l} \texttt{vcard} \mapsto \left[\texttt{Davide} \mapsto \left[\begin{array}{l} \texttt{org} \mapsto \left[\texttt{Bologna} \mapsto [\\ \texttt{email} \mapsto \left[\texttt{Davide@cs.unibo} \mapsto [\end{array} \right. \end{array} \right. \right. \\ \texttt{vcard} \mapsto \left[\texttt{Rocco} \mapsto \left[\begin{array}{l} \texttt{org} \mapsto \left[\texttt{Edinburgh} \mapsto [\\ \texttt{email} \mapsto \left[\texttt{denocola@dcs.ed} \mapsto [\end{array} \right. \end{array} \right. \right. \end{array} \right. \right.$$

Transforming Away Ordering

$$\left\{ \texttt{xcard} \mapsto \left[\begin{array}{l} \texttt{Davide} \mapsto \left[\begin{array}{l} \texttt{org} \mapsto \left[\texttt{Bologna} \mapsto [\right. \\ \texttt{email} \mapsto \left[\texttt{Davide@cs.unibo} \mapsto [\right. \end{array} \right. \\ \texttt{Rocco} \mapsto \left[\begin{array}{l} \texttt{org} \mapsto \left[\texttt{Edinburgh} \mapsto [\right. \\ \texttt{email} \mapsto \left[\texttt{denocola@dcs.ed} \mapsto [\right. \end{array} \right. \end{array} \right. \right.$$

Transforming Away Ordering

$$\left\{ \texttt{xcard} \mapsto \left\{ \begin{array}{l} \texttt{Davide} \mapsto \left[\begin{array}{l} \texttt{org} \mapsto \left[\texttt{Bologna} \mapsto [\right. \\ \texttt{email} \mapsto \left[\texttt{Davide@cs.unibo} \mapsto [\right. \end{array} \right. \\ \texttt{Rocco} \mapsto \left[\begin{array}{l} \texttt{org} \mapsto \left[\texttt{Edinburgh} \mapsto [\right. \\ \texttt{email} \mapsto \left[\texttt{denocola@dcs.ed} \mapsto [\right. \end{array} \right. \end{array} \right. \right.$$

Transforming Away Ordering

$$\left\{\begin{array}{l} \texttt{Davide} \mapsto \left[\begin{array}{l} \texttt{org} \mapsto \left[\texttt{Bologna} \mapsto [\right. \\ \texttt{email} \mapsto \left[\texttt{Davide@cs.unibo} \mapsto [\right. \end{array}\right. \\ \texttt{Rocco} \mapsto \left[\begin{array}{l} \texttt{org} \mapsto \left[\texttt{Edinburgh} \mapsto [\right. \\ \texttt{email} \mapsto \left[\texttt{denocola@dcs.ed} \mapsto [\right. \end{array}\right. \end{array}\right.$$

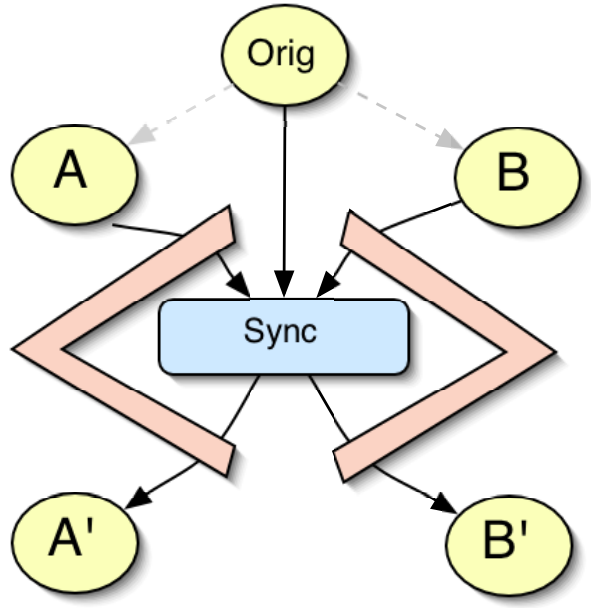
Transforming Away Ordering

$$
\left\{
\begin{array}{l}
\texttt{Davide} \mapsto \left\{
\begin{array}{l}
\texttt{org} \mapsto \left\{\texttt{Bologna} \mapsto \{\right. \\
\texttt{email} \mapsto \left\{\texttt{Davide@cs.unibo} \mapsto \{\right.
\end{array}
\right. \\
\\
\texttt{Rocco} \mapsto \left\{
\begin{array}{l}
\texttt{org} \mapsto \left\{\texttt{Edinburgh} \mapsto \{\right. \\
\texttt{email} \mapsto \left\{\texttt{denocola@dcs.ed} \mapsto \{\right.
\end{array}
\right.
\end{array}
\right.
$$

Lenses

Of course, after synchronization, we need our data back in its original concrete form.

I.e., the reorganization transformation must be "wrapped around" both sides of the core synchronization engine. These bi-directional transformations are called lenses.



A Programming Language for Lenses

Harmony includes a domain-specific programming language in which all well-typed expressions denote "well-behaved" lenses. [See our POPL '05 paper.]

For this talk, though, let's concentrate on the synchronization engine.

Conflicts

Conflicts

We have seen that, by changing representation, we can eliminate a variety of spurious conflicts.

However...

Some Conflicts are Inevitable

The essence of optimistic replication (and the reason that it is called optimistic) is that replicas can sometimes be updated in truly conflicting ways.

Some Conflicts are Inevitable

The essence of optimistic replication (and the reason that it is called optimistic) is that replicas can sometimes be updated in truly conflicting ways.

In the present setting, a conflict occurs when some subtree of one replica is deleted and the corresponding subtree of the other replica is modified.

Dealing with Conflicts

When a conflict occurs, we have an uncomfortable choice:

1. give up persistence

   $\Longrightarrow$ synchronization "backs out" changes made by the user at one of the replicas

2. give up convergence

   $\Longrightarrow$ synchronization leaves the replicas unequal

For a state-based synchronizer like Harmony, the second seems more natural.

Synchronization Algorithm

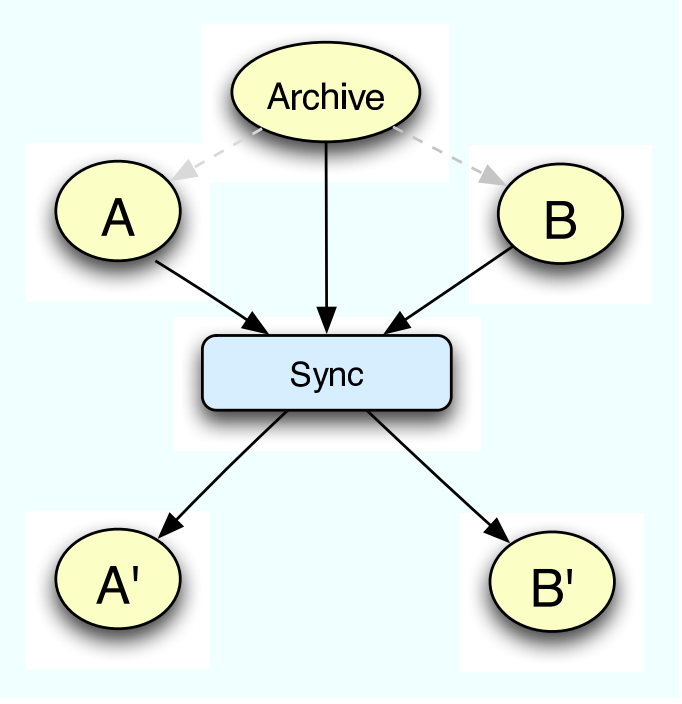
Notation

- names, ranged over by k
- a path p is a sequence of names
- a tree is a finite function from names to trees
- the contents of a tree a at some name k, written $a(k)$, is either a tree or $\bot$
- write $\mathcal{T}$ for the set of all trees
- $\mathcal{T}_\bot = \mathcal{T} \cup \{\bot\}$

The Archive

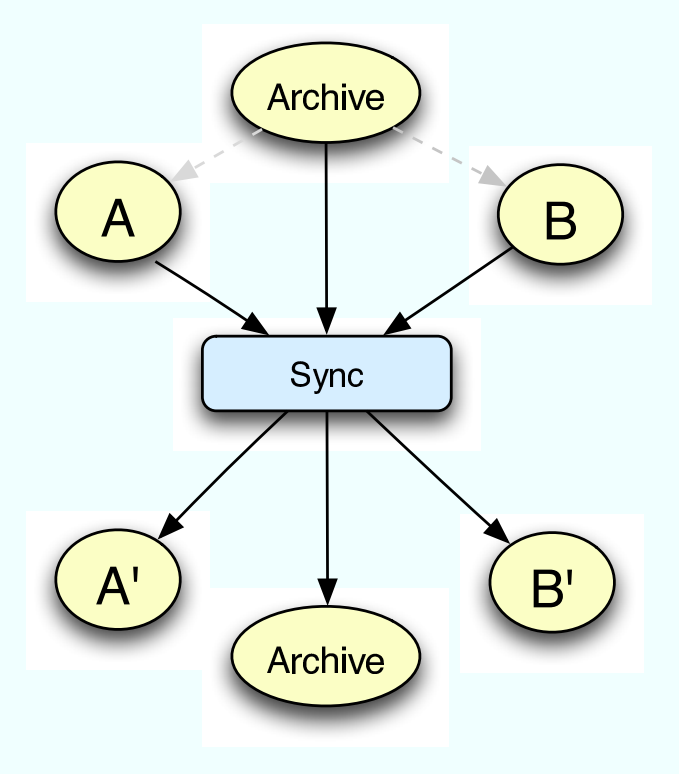
We have seen that the synchronizer needs to be told the "last synchronized state" of the two replicas.

We call this the archive.



The Archive

In order to synchronize repeatedly, the archive must also be an output of the synchronization algorithm.



Synchronization Algorithm (First Try)

$$sync \quad \in \quad (\mathcal{T}_{\perp\mathcal{X}} \times \mathcal{T}_{\perp} \times \mathcal{T}_{\perp}) \longrightarrow (\mathcal{T}_{\perp\mathcal{X}} \times \mathcal{T}_{\perp} \times \mathcal{T}_{\perp})$$

$sync(o, a, b) =$

if $a = b$ then(a, a, b)	- equal replicas: done
else if $a = o$ then (b, b, b)	- no change to a: propagate b
else if $b = o$ then (a, a, a)	- no change to b: propagate a
else if $o = \mathcal{X}$ then (o, a, b)	- unresolved conflict
else if $a = \perp$ then $(\mathcal{X}, a, b)$	- delete/modify conflict
else if $b = \perp$ then $(\mathcal{X}, a, b)$	- delete/modify conflict
else	- proceed recursively...

let $(o'(k), a'(k), b'(k)) = sync(o(k), a(k), b(k))$

$\forall k \in dom(a) \cup dom(b)$

(o', a', b')

More Difficulties

Our current synchronization algorithm is a bit too eager: it will often merge changes in ways that yield mangled results.

$$o = \{\texttt{org} \mapsto \{\texttt{Edinburgh} \mapsto \{$$

$$a = \{\texttt{org} \mapsto \{\texttt{INRIA} \mapsto \{$$

$$b = \{\texttt{org} \mapsto \{\texttt{Bologna} \mapsto \{$$

$$a' = b' = \{\texttt{org} \mapsto \begin{cases}\texttt{INRIA} \mapsto \{ \\ \texttt{Bologna} \mapsto \{\end{cases}$$

More Difficulties

Similarly, suppose we want every address book entry to contain either an email address or an organization.

- start with a record containing both `email` and `org`
- delete `email` in one replica
- delete `org` in the other replica
- note that all three variants satisfy
- now synchronize...

More Difficulties

Similarly, suppose we want every address book entry to contain either an email address or an organization.

- start with a record containing both `email` and `org`
- delete `email` in one replica
- delete `org` in the other replica
- note that all three variants satisfy
- now synchronize...
- both deletions get propagated, yielding an ill-formed result.

The Role of Schemas

Schema-Aware Synchronization

A simple way to prevent the synchronizer from returning ill-formed structures is to tell it not to!

Synchronization algorithm should...

- check output replicas against intended schema
- signal a conflict if either check fails

Formally...

A Simple Schema-Aware Synchronizer

$bettersync(S, o, a, b) =$

let $(o', a', b') = sync(o, a, b)$

in if $(a' \notin S)$ **or** $(b' \notin S)$

then $(\mathcal{X}, a, b)$ – schema conflict

else (o', a', b')

Throwing out Baby with Bathwater

This is too coarse-grained: A conflict anywhere will lead to a synchronization failure everywhere!

We want something more local...

Final Synchronization Algorithm

$sync(S, o, a, b) =$

if $a = b$ then (a, a, b) - equal replicas: done

else if $a = o$ then (b, b, b) - no change to a: propagate b

else if $b = o$ then (a, a, a) - no change to b: propagate a

else if $o = \mathcal{X}$ then (o, a, b) - unresolved conflict

else if $a = \perp$ then $(\mathcal{X}, a, b)$ - delete/modify conflict

else if $b = \perp$ then $(\mathcal{X}, a, b)$ - delete/modify conflict

else - proceed recursively...

let $(o'(k), a'(k), b'(k)) = sync(S(k), o(k), a(k), b(k))$

$\forall k \in dom(a) \cup dom(b)$

in if $(a' \notin S)$ or $(b' \notin S)$

then $(\mathcal{X}, a, b)$ - schema conflict

else (o', a', b')

Final Synchronization Algorithm

$sync(S, o, a, b) =$

if $a = b$ then (a, a, b) - equal replicas: done

else if $a = o$ then (b, b, b) - no change to a: propagate b

else if $b = o$ then (a, a, a) - no change to b: propagate a

else if $o = \mathcal{X}$ then (o, a, b) - unresolved conflict

else if $a = \perp$ then $(\mathcal{X}, a, b)$ - delete/modify conflict

else if $b = \perp$ then $(\mathcal{X}, a, b)$ - delete/modify conflict

else - proceed recursively...

let $(o'(k), a'(k), b'(k)) = sync(S(k), o(k), a(k), b(k))$

$\forall k \in dom(a) \cup dom(b)$

in if $(a' \notin S)$ or $(b' \notin S)$

then $(\mathcal{X}, a, b)$ - schema conflict

else (o', a', b')

Path Consistency

To ensure that we can "project" a schema one a given name, we need to consider only schemas of a restricted form.

Definition: A schema S is **path consistent** iff, for all trees $t, t' \in S$ and paths p, we have

$$t(p) \neq \bot \wedge t'(p) \neq \bot \implies t[p \mapsto t'(p)] \in S,$$

where $t[p \mapsto t'(p)]$ is the tree obtained by replacing the subtree of t at p by the corresponding subtree of t'.

Path Consistency

Path-consistent schemas are a "semantic analog" of single-type tree grammars used in W3C Schema.

The are expressive enough to describe a wide range of examples.

Specification of Synchronization

Specification

A good synchronizer should...

1. Never "back out" changes
2. Never "make up" contents
3. Stop at conflicting paths (leaving replicas in their current states)
4. Always leave the replicas in a well-typed form

safety conditions

5 Propagate as many changes as possible without violating above rules

maximality condition

The (Theoretical) Punchline

Theorem: The final (schema-aware) synchronization algorithm satisfies all these conditions.

Proof: [See paper.]

Related Work

Related Projects

- IceCube
  - ongoing project at MSR (Shapiro, Rowstrom, Kemmarek, ...)
  - operation-based synchronization middleware
  - sophisticated algorithms for finding "best" merges of operation sequences from different replicas
- Bayou
  - late '90s project at Xerox PARC (Edwards, Mynatt, Petersen, Spreitzer, Terry, Theimer, ...)
  - operation-based
  - not as flexible as IceCube, but addressed distribution / scale issues very seriously
- Unison
  - late '90s project at Penn (Jim, Pierce, Vouillon)
  - state-based file synchronizer
  - formal specification similar to Harmony's

Related work

- Ramsey and Csirmaz
  - Careful algebraic specification of a file synchronizer similar to Unison, in an operation-based style
- LibreSource
  - current project at INRIA Lorraine [Molli, Oster, Skaf-Molli, Imine, etc.]
  - basic idea: operation transform

    Define, for each pair of operations, op_1 and op_2, a transformed version of op_1, written $T(op_1, op_2)$, that achieves "the same effect" as op_1 but makes sense in a context where op_2 has been executed.

Finishing Up...

Harmony Status

- Core implementation and several demos running
  - universal bookmark synchronizer (Internet Explorer, Mozilla, Safari, ...)
  - synchronizer for calendars in several formats
  - (simple) structured text file synchronizer
  - bibtex synchronizer
- 2 users :-)

Planned Demos

- address books
- preference files
- diagrams
- biological database annotations
- slide presentations (Powerpoint, Keynote)
- richer structured documents (Word, LaTeX, DocBook, etc.)
- What would you like to synchronize??

Ongoing Work

- public release
  [sometime between Tiger and Longhorn :-)]
- multi-replica synchronization
  [some preliminary work is described in a recent draft paper]
- beyond tree-structured data
  - hybridizing local and global synchronization techniques to handle list-structured data
  - synchronizing dags [with Sanjeev Khanna and Alan Schmitt]
  - synchronizing relational data, sets, bags, etc., etc.

Acknowledgments

Main collaborators on this work: Nate Foster, Michael Greenwald, and Alan Schmitt

Other Harmony contributors: Malo Denielou, Owen Gunden, Sanjeev Khanna, Christian Kirkegaard, Keshav Kunal, Jonathan Moore, and Zhe Yang

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