Synchronize Globally, Compute Locally

Benjamin C. Pierce

University of Pennsylvania

Global Computing

- Large research area, tying together many (old and new) topics
  - mobile agents
  - process migration and communication infrastructure
  - distributed algorithms
  - failure models
  - security, authentication, and trust issues
  - etc.

Global Computing

- Large research area, tying together many (old and new) topics
  - mobile agents
  - process migration and communication infrastructure
  - distributed algorithms
  - failure models
  - security, authentication, and trust issues
  - etc.

- Major interest in migratory computation

Global Computing

- Large research area, tying together many (old and new) topics
  - mobile agents
  - process migration and communication infrastructure
  - distributed algorithms
  - failure models
  - security, authentication, and trust issues
  - etc.
- Major interest in migratory computation
- What about migratory **data**?

Data in Global Networks

- Some data structures are so tiny and/or transient that they can simply be carried along with computations as they move
- Others are so enormous that it only makes sense to have one copy
- Between these two extremes are a wide range of structures for which some sort of replication is desirable (e.g., to increase availability)

Approaches to Replication

Pessimistic schemes:

- Replicate data at many hosts
- Propagate updates to all replicas immediately

Approaches to Replication

Pessimistic schemes:

- Replicate data at many hosts
- Propagate updates to all replicas immediately

$\Longrightarrow$ Costly agreement protocols

Approaches to Replication

Pessimistic schemes:

- Replicate data at many hosts
- Propagate updates to all replicas immediately

$\Longrightarrow$ Costly agreement protocols

Problem: Scale!

Approaches to Replication

Mirroring:

- Read/write copies of data on just one host or replicated (pessimistically) on a small number of hosts
- Read-only copies "mirrored" to many more hosts
- Changes to master copies propagate to mirrors "every so often"

Improves scalability of pessimistic replication schemes, at some cost in data freshness. (This is acceptable for many applications, e.g., web caching.)

Approaches to Replication

Mirroring:

- Read/write copies of data on just one host or replicated (pessimistically) on a small number of hosts
- Read-only copies "mirrored" to many more hosts
- Changes to master copies propagate to mirrors "every so often"

Improves scalability of pessimistic replication schemes, at some cost in data freshness. (This is acceptable for many applications, e.g., web caching.)

Problem: Availability/scalability for **updates**

Approaches to Replication

Optimistic schemes:

- Many copies of data
- Any host may update its own copy at any time
- Hosts occasionally compare their states and propagate updates as necessary

Approaches to Replication

Optimistic schemes:

- Many copies of data
- Any host may update its own copy at any time
- Hosts occasionally compare their states and propagate updates as necessary

Optimistic strategies are
the only option in many situations.

But they raise tricky issues...

Issues in Optimistic Replication

- **Algorithmic issues**
  - making sure that everybody hears about updates
  - ...without swamping the network

  Challenging if there are many copies and/or frequent updates!

  $\Longrightarrow$ Epidemic and gossip algorithms, version vectors, etc.

- **Semantic issues**
  - combining "parallel" updates to the same data
  - recognizing and dealing with conflicts

  Need to offer users (and application programmers) a clean model of "what [/where/when] their data is"

Issues in Optimistic Replication

- **Algorithmic issues**
  - making sure that everybody hears about updates
  - ...without swamping the network

  Challenging if there are many copies and/or frequent updates!

  $\Longrightarrow$ Epidemic and gossip algorithms, version vectors, etc.

- **Semantic issues**
  - combining “parallel” updates to the same data
  - recognizing and dealing with conflicts

  Need to offer users (and application programmers) a clean model of “what [/where/when] their data is”

  $\Longrightarrow$ **Synchronization**

Synchronization is Everywhere

Synchronization technologies are commonplace in the systems that we use today...

- PDA hotsync utilities
- distributed filesystems
- distributed databases
- group calendar managers
- version control systems
- autoteller machines
- ad hoc tools for scientific databases
- etc., etc.

...and are becoming even more so.

(Unison Demo)

Synchronization is Challenging

Synchronization may appear straightforward... but it's not!

Some Basic Issues

- What is a **conflict**?

  Easy example: Suppose we have a replicated directory structure.

  - in one replica, we modify a file;
  - in another replica, we delete it.

  If we now synchronize, we clearly have a conflict.

Some Basic Issues

- What is a **conflict**?

  Easy example: Suppose we have a replicated directory structure.

  - in one replica, we modify a file;
  - in another replica, we delete it.

  If we now synchronize, we clearly have a conflict.

  Trickier example:

  - in one replica, we modify a file;
  - in another replica, we delete the directory that contains it.

  What, exactly, do we expect to happen if we now synchronize these replicas?

Some Basic Issues

- What is a change?
  - E.g., if we write a new copy of a file that happens to be exactly the same as the old one, should this count as an update to the file?
  - If we (simultaneouly) make a different update to another replica, should synchronization propagate the second change or signal a conflict?

Some Basic Issues

- Should synchronization be considered as a matter of reconciling the current states of the replicas, or is it better to consider it as a problem of merging the logs of the operations that got us into these states?

Some Basic Issues

- If synchronization is interrupted (e.g., by host or network failure), what states might the replicas be left in?

Some Basic Issues

- How can all this be described precisely, in terms that make sense to both users and implementors?

Some Basic Issues

- How can all this be described precisely, in terms that make sense to both users and implementors?

$\Longrightarrow$ Need to look for **principles**...
("semantics of synchronization"?)

Unison and Harmony

The **Unison** project has focused on developing principles (and "principled tools") for the special case of **file synchronization**.

Our new project, **Harmony**, aims to develop principles (and tools) for the more general problem of **XML synchronization**.

The Unison Project

6-year project, mostly at University of Pennsylvania

Personnel:

Benjamin 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.

Unison's Contributions

- Rigorous mathematical specification
- Elegant reference implementation with formal proof of correctness
- Robust industrial-strength implementation
  - developed in parallel with specification
  - significant user community (several thousand users [and growing!]; included in SuSe, Debian, and Mandrake Linux distributions; etc.)
  - runs on (and between!) Windows and Unix systems

States vs. logs

Recall our earlier question about whether synchronization should focus on reconciling the states of replicas or the logs of individual updates that produced them.

This is a fundamental choice, both at the level of specification and at the level of engineering...

States vs. logs

- Log-based synchronizers are given complete traces of all the actions on the filesystems.

  This approach is more powerful (in principle). But...

  - Requires tight integration with operating system
  - Logs can become huge
  - Merging even modest-sized logs poses challenging combinatorial problems [cf. IceCube project]

- State-based synchronizers use only static information: the current state of the replicas and the last-synchronized state.

  This makes them (much) easier to implement.

Unison and Harmony are both state-based.

Principles

Definitions:

- user change = a path whose contents are different from when this path was last synchronized
- conflict = path whose contents have changed in one replica and whose contents or descendents have changed in the other

Principles

Definitions:

- user change = a path whose contents are different from when this path was last synchronized
- conflict = path whose contents have changed in one replica and whose contents or descendents have changed in the other

Specification:

1. Never overwrite user changes
2. Propagate only user changes
3. Do nothing on (or below) conflicting paths
4. Propagate as many changes as possible (subject to 1–3)

Principles

Definitions:

- user change = a path whose contents are different from when this path was last synchronized
- conflict = path whose contents have changed in one replica and whose contents or descendents have changed in the other

Specification:

1. Never overwrite user changes
2. Propagate only user changes   Duh?
3. Do nothing on (or below) conflicting paths
4. Propagate as many changes as possible (subject to 1–3)

Principles $\longrightarrow$ Practice

Theorem: The four clauses above completely determine the behavior of a synchronizer.

That is, they constitute a complete description of what an implementation must achieve — a rigorous contract between users and implementors.

Our implementation takes these principles extremely seriously, which requires pretty hard work in some places,

Its resulting robustness and predictability are very popular with users!

Going Further...

Specifying and building Unison has led to some clear and natural principles of synchronization for the particular case of filesystems.

The same principles should apply (straightforwardly) to other kinds of “filesystem-like” data — i.e., any data that can naturally be represented by unordered labeled trees.

But... there are lots of other kinds of data in the world.

Going Further...

Specifying and building Unison has led to some clear and natural principles of synchronization for the particular case of filesystems.

The same principles should apply (straightforwardly) to other kinds of “filesystem-like” data — i.e., any data that can naturally be represented by unordered labeled trees.

But... there are lots of other kinds of data in the world.

Challenge: Can we develop similar principles (and tools) for a broader class of structures?

E.g., can we build an XML synchronizer along these lines?

The Harmony Project

New project (with major funding from National Science Foundation)

Personnel:

- Michael Greenwald
- Sanjeev Khanna
- Jon Moore
- Benjamin Pierce
- Zhe Yang

(all at Penn)

Key problem: Ordered data

In many XML structures (bookmarks, textual documents, etc.), order is significant. This makes synchronization much trickier.

Example: Suppose we replicate a list of integers:

1 2 3 4 5 6 7 8 9

In one replica, we exchange 4 and 5:

1 2 3 5 4 6 7 8 9

In another replica, we move 1 to the end:

2 3 4 5 6 7 8 9 1

If we now synchronize these replicas, what should happen?

How do we design an algorithm that achieves this?

Proposal

A possible architecture for a generic synchronization framework:

- Small collection of elementary structures for which we can give a precise and natural specification for synchronization — in particular, where alignment is clear.
- Tools that construct views of “arbitrary data structures” as elementary structures (and interpret changes to those views as updates to the underlying structures).

Parting Thoughts...

Synchronization is a rich area for further research, full of challenges and puzzles.

Focusing on synchronization suggests an alternative perspective for global computing...

- data (not computation) is “global”
- updates “migrate” between replicas
- most computation is local

Synchronize globally, compute locally

Related Projects

Many implementations:

- Distributed file systems [Coda, Ficus, etc., etc.]
- File synchronizers [MS Briefcase, Puma IntelliSync, Rumor, Reconcile, etc.]
- Synchronization middleware [Bayou, ...]
- Version control systems [CVS, PRCS, ...], program integration tools [Reps&Horwitz, ...], etc., etc.

Related Projects

Some specifications:

- Norman Ramsey [Harvard] — algebraic specification of a unison-like file synchronizer
- IceCube [Marc Shapiro & co, MSR-UK] — trace-based specifications of a more general middleware layer

Want to play?...

Papers on Unison (as well as full sources, docs, and pre-built binaries) can be found here:

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

Preliminary design sketches for Harmony can be found here:

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

Extra slides

Log- vs. state-based

Log-based synchronizers have more information to work with, so they can, in principle, make better decisions.

However:

- in general, we want to be able to synchronize many replicas, some of which may go off-line for a long time

  $\longrightarrow$ we might have to keep huge logs (everywhere)

- even if the logs are reasonable size, making good use of all this information poses hard combinatorial problems (cf. IceCube)

- obtaining logs requires tight integration between the synchronizer and the rest of the system (e.g., it must be built into the filesystem); state-based synchronizers are generally lighter-weight and more independent

Example

- start with file f in both replicas containing x; synchronize
- in both replicas, change the contents of file f to y
- next, in one replica, change the contents of f to z
- now re-synchronize
  - state-based synchronizer sees new contents y in one replica and z in the other: conflict
  - log-based synchronizer can see that z is the desired final state

Track I: Enriching the elementary structures

Understood:

- tree structures
- sets, bags
- flat databases (relations)
- counters
- simple ordered structures (tuples)

Not understood:

- more general ordered structures
  (e.g. lists, documents, priority rankings)

Track I: Enriching the elementary structures

Understood:

- tree structures
- sets, bags
- flat databases (relations)
- counters
- simple ordered structures (tuples)

Not understood:

- more general ordered structures $\Longleftarrow$ start here

  (e.g. lists, documents, priority rankings)

- graphs (e.g., filesystems!)
- programs
- etc., etc.

- graphs (e.g., filesystems!)
- programs
- etc., etc.

Track II: Constructing Views

1. Fix a very simple set of elementary structures

2. Pick a real-world application domain involving nontrivial – but not horrendously complicated – data structures

3. Try (by experimentation) to find a good way of constructing views of these application data structures as elementary structures.

Track II: Constructing Views

1. Fix a very simple set of elementary structures... unordered, edge-labeled trees (i.e., filesystems!)
2. Pick a real-world application domain involving nontrivial – but not horrendously complicated – data structures
3. Try (by experimentation) to find a good way of constructing views of these application data structures as elementary structures.

Track II: Constructing Views

1. Fix a very simple set of elementary structures... unordered, edge-labeled trees (i.e., filesystems!)

2. Pick a real-world application domain involving nontrivial – but not horrendously complicated – data structures... PalmOS databases

3. Try (by experimentation) to find a good way of constructing views of these application data structures as elementary structures.

Example

[picture]

Current status

Running prototype...

- supporting address books, datebooks, memos, and mail
- constructed entirely using views and filters
- nonzero user community :-)

Current status

Running prototype...

- supporting address books, datebooks, memos, and mail
- constructed entirely using views and filters
- nonzero user community :-)

More questions than answers...

- what are (the types of) views / filters? what is their ”abstract nature” (i.e., what sorts of mathematical objects help us conceptualize them?)
- what is a good (”complete”, etc.) set of basic filter combinators?
- can filters be generated automatically from readily available

information like schemas?

- what algebraic laws do filters obey (useful for optimization)?
- can we design a domain-specific language for programming filters?

information like schemas?

- what algebraic laws do filters obey (useful for optimization)?
- can we design a domain-specific language for programming filters?