

~~Principle of Synchronisation~~

Benjamin C. Pierce
University of Pennsylvania

Workshop on Cloud Services for File Synchronisation and Sharing
CERN, November 2014

Mysteries of Dropbox

Benjamin Pierce

University of Pennsylvania

John Hughes

Thomas Arts

Quviq

Workshop on Cloud Services for File Synchronisation and Sharing

CERN, November 2014

We're here because we care about
synchronization!

Actually, a lot of people care about
synchronization



~200M

And they care a lot

So...

Is yours correct?

What does that even mean??

Goals

- Ultimate goal: Answer this question rigorously
 - i.e., give a *complete* and *precise* definition of how a synchronization service should behave (from the point of view of users, not protocols)
 - i.e., write down a *formal specification* that, for each observed interaction with a synchronization service (involving several replicas being read and written over time), says whether or not it is OK
- Goal for this talk:
 - Report on baby steps in this direction



A Little History

- **Unison** is the only synchronizer based on a formal specification (AFAIK)
- Main designers
 - Trevor Jim, Jerome Vouillon, and BCP
- First version distributed in 1998
 - Earlier prototypes going back to 1995
 - Still widely used
- Open-source, multi-platform (Windows, OSX, Linux)
- *Very* few reported bugs with potential to lose data

But the world has changed...

Synchronization tools: (bidirectional, sync operations explicit)

- Unison, ...



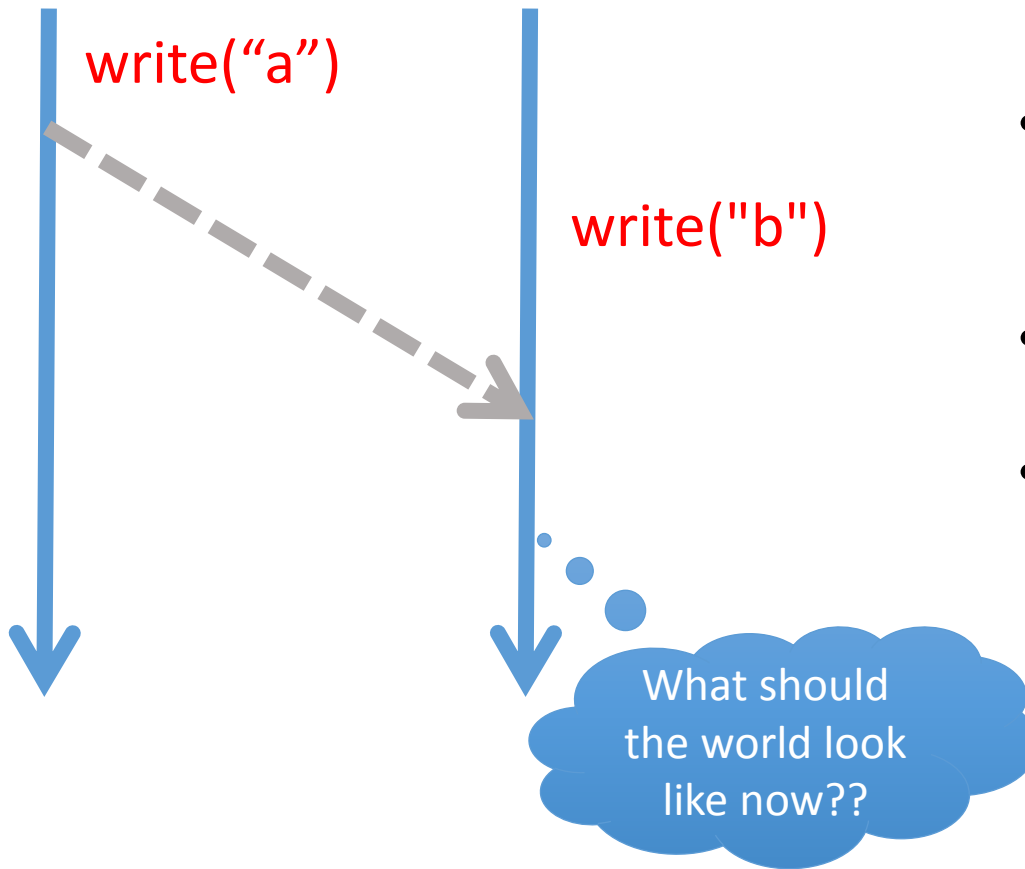
Synchronization services: (multi-directional, sync in background)

- Central server (“cloud-based”) • • • concentrate on these
 - Dropbox, Google Drive, OneDrive, Owncloud, SpiderOak, Sugarsync, Box.net, Seafile, Pulse, Wuala, Teamdrive, Cloudme, Cx, Amazon cloud service, ...
 - Also distributed filesystems (Coda, GFS, ...)
- Peer to Peer
 - Bittorrentsync

Challenges

- Syncing in background produces **inherently nondeterministic** behaviour
 - Our specification will need to specify *sets* of legal behaviors
- Similar to **weak memory models** for multi-core processors...
- But **harder!**
 - directory structure
 - deletion
 - conflicts
 - ...

One challenge: Conflicts



- Unison asks the user to resolve
- Modern systems just *do something*
- e.g., choose "earlier" value, create a "conflict file" containing other value

Critical to do the right something!

Another challenge:

How can we test that our specification is accurate??

This is hard!

So...

Don't write tests!

Generate them

QuickCheck



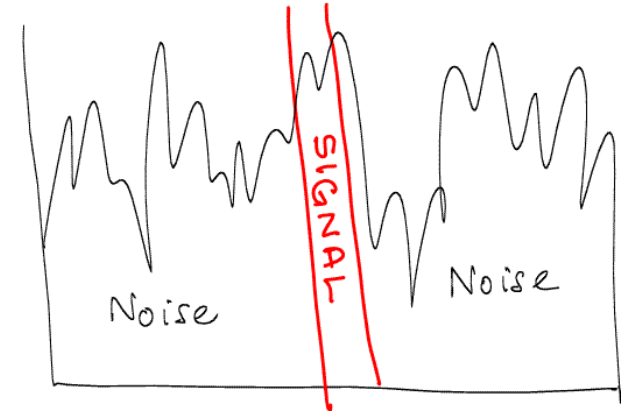
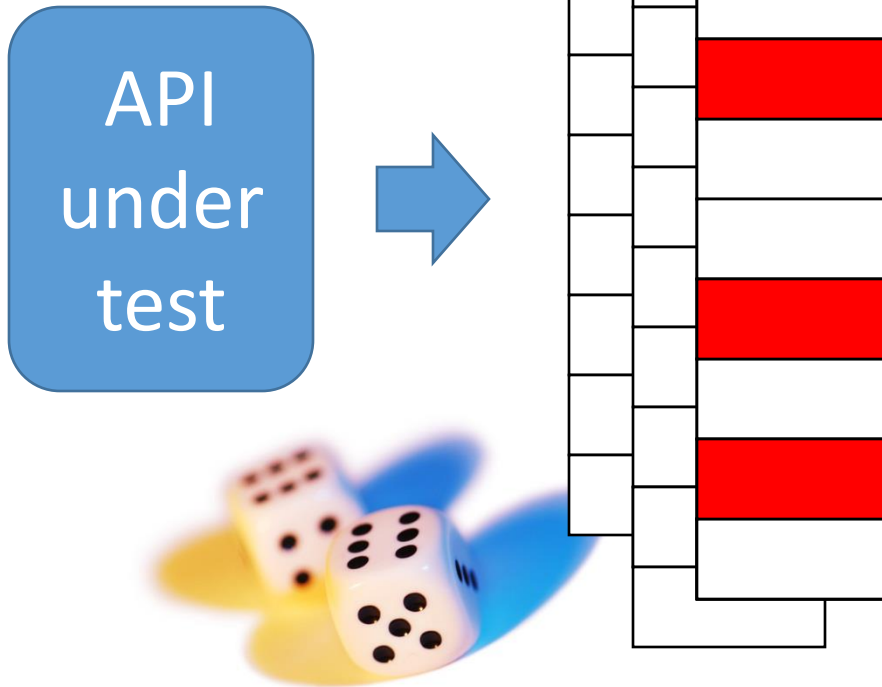
1999—invented by Koen Claessen and John Hughes,
for Haskell

2006—Quviq founded, marketing Erlang version

Many extensions

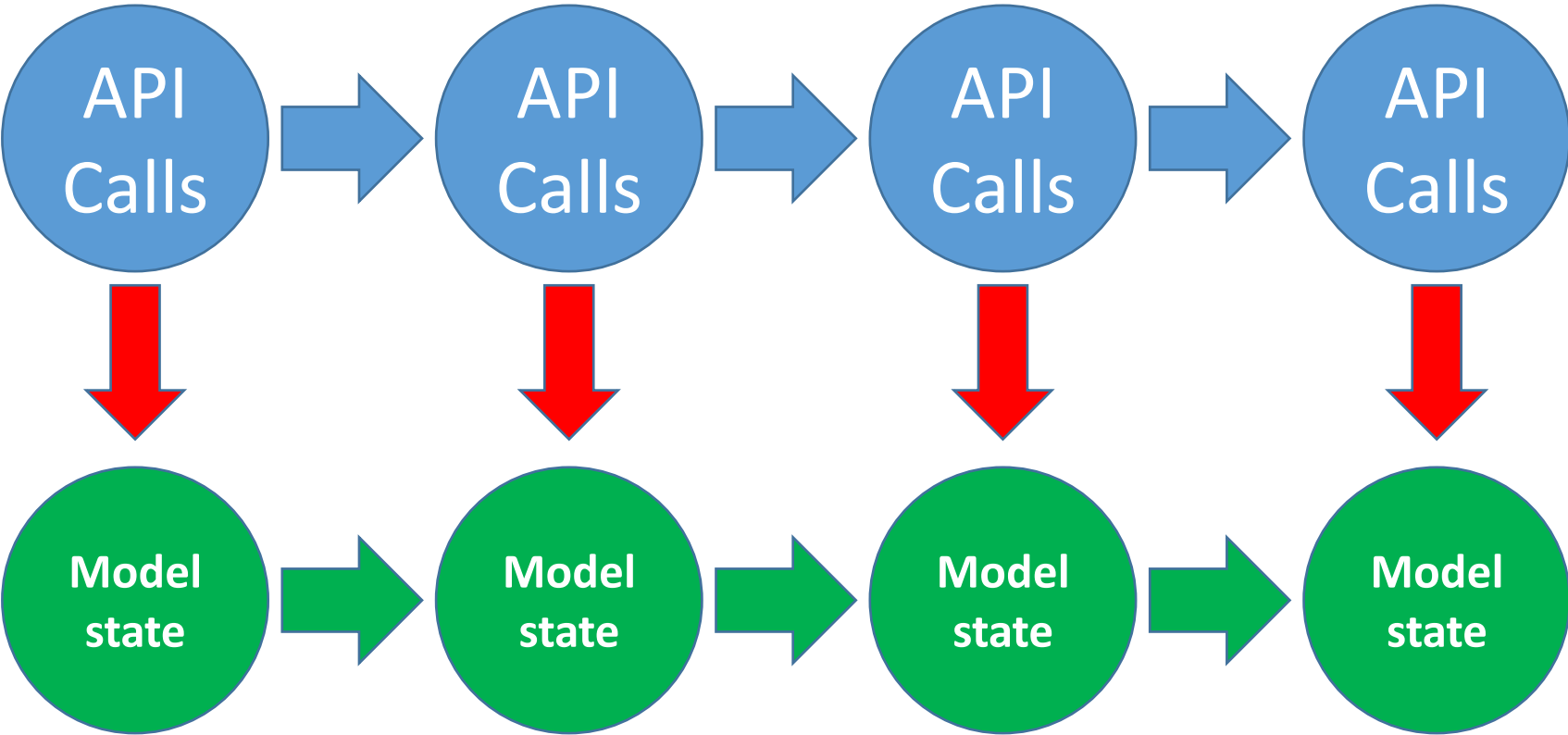
Finding deep bugs for Ericsson, Volvo Cars, Basho,
etc...

QuickCheck

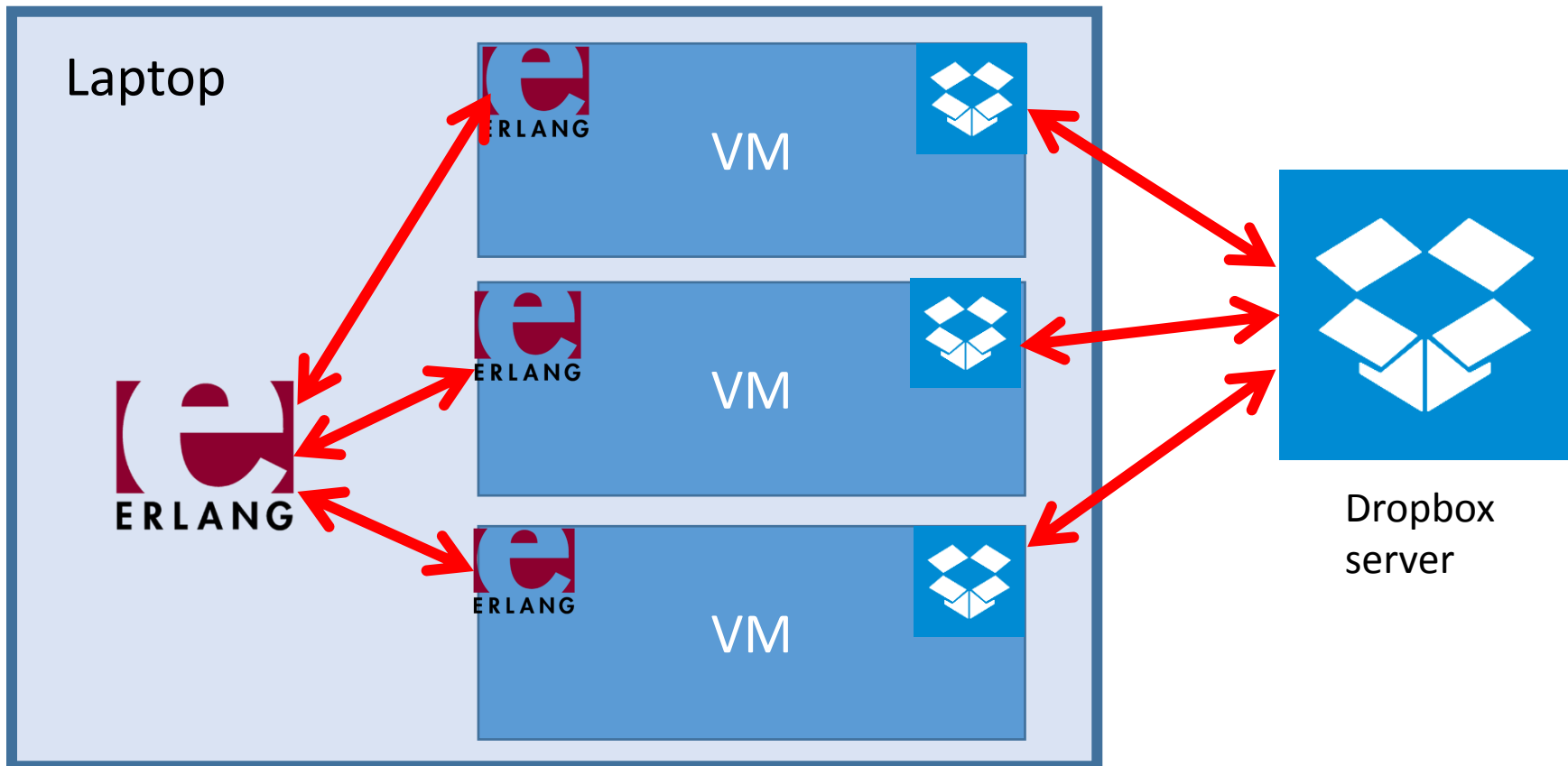


A minimal failing example

State Machine Models



Test Setup



Challenges of testing

- Uniformity
 - *Many* synchronization services
 - Want a single specification and test framework that applies to multiple systems
- File synchronizers are SLOOOOWWWW!!!
 - Exponential back-off
 - No such thing as waiting "long enough"
 - Tests must adapt to the speed of the synchronizer
- Interference between tests
 - Reads/writes/deletes in the same directory
 - → Isolate tests by using many directories
 - Test setup:
 - Delete old directories, and wait
 - Create new directories, and wait...

and wait... and wait...

A small simplification

Real-world filesystems are pretty complicated

So let's start with something a little simpler...

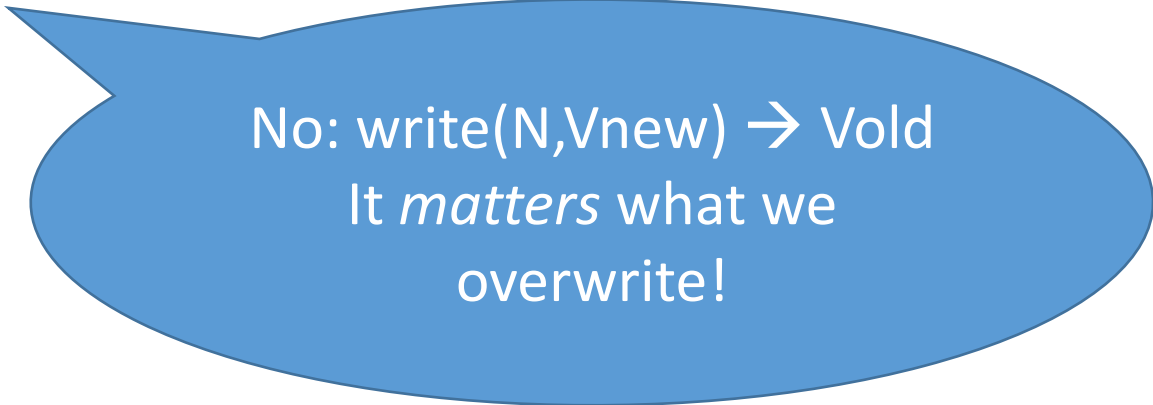
“Filesystem” = 1 file
Operations: read, write
(and, later, delete)

Tests are sequences of commands

- read(Node)
- write(Node,V)
- sleep
- ~~synchronize?~~
- stabilize
 - Wait for *value* observed on each node to be the same
(Also *conflict files*)

Observations

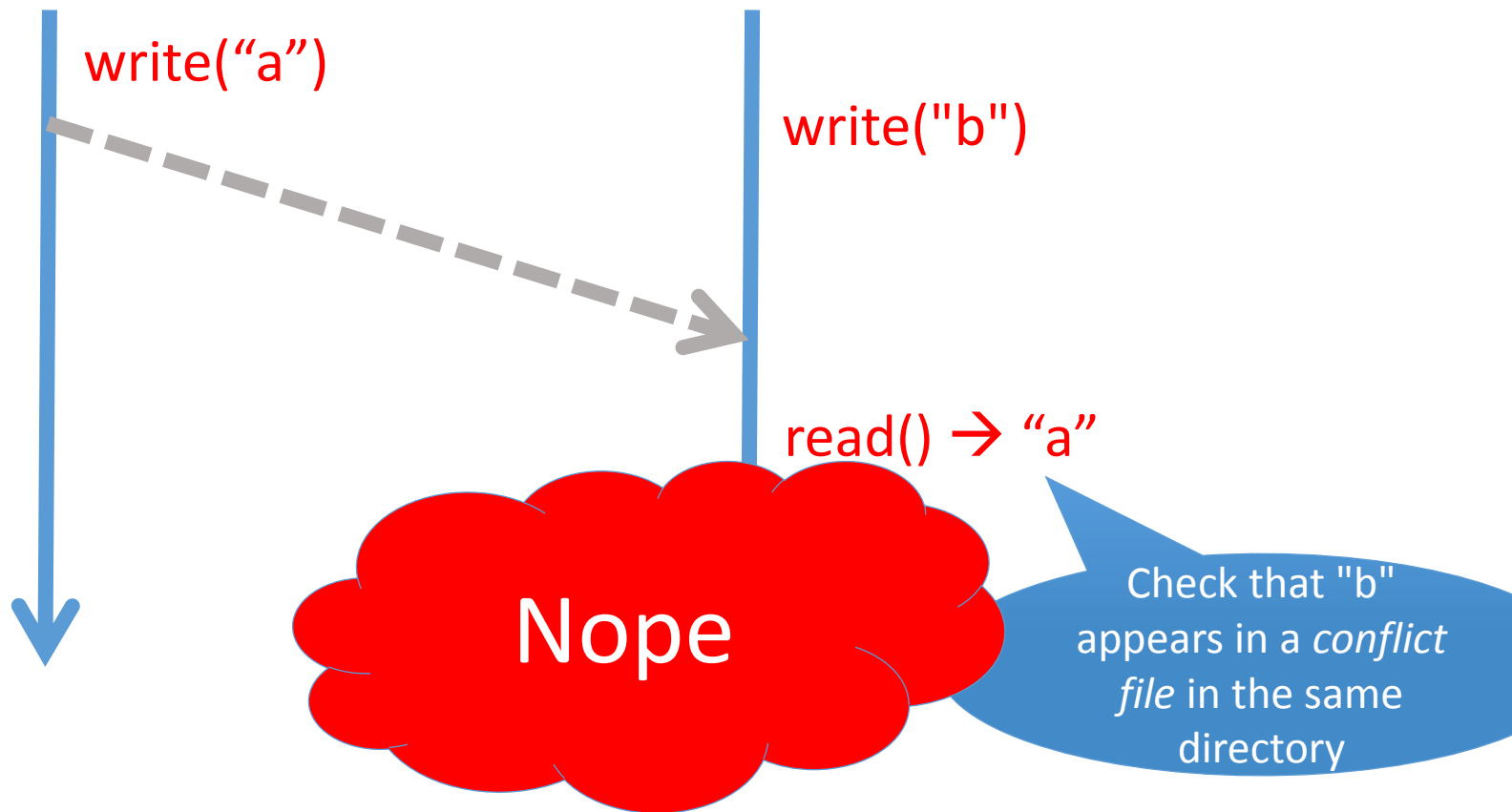
- Made dynamically; the specification says which observation sequences are **valid**
- What do we observe?
 - $\text{read}(N) \rightarrow V$
 - $\text{write}(N, V_{\text{new}}) \rightarrow ()$
 - Conflicts??



No: $\text{write}(N, V_{\text{new}}) \rightarrow \text{Void}$
It *matters* what we
overwrite!

Observing conflicts?

- First try: when a write creates a conflict, check that one of the conflicting values appears in a conflict file

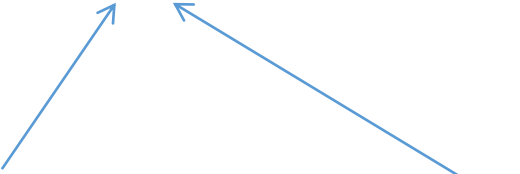


Observing conflicts... eventually!

- New observation:

- stabilize() → (V,C)

stable value
(same everywhere)



set of values found in
conflict files
(same everywhere)

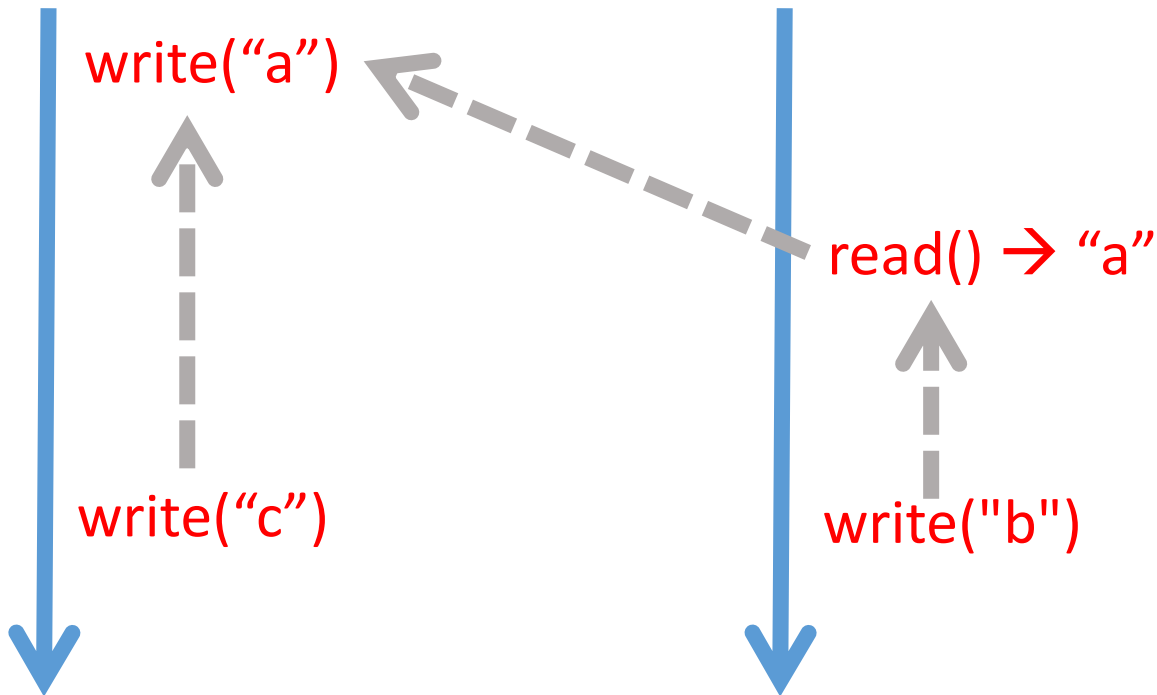
Now...

What should our
specification look like?

A dead end

But what about
repeated values?
E.g. deletion?

- Compute a "happened before" relation (*à la* weak memory models) and express correctness in terms of that...



A Better Idea

- Model the whole system state *including the server*
- Update the state after each observation
- Add "conjectured actions" to the observed ones
 - up(N) node N uploads its value to the server
 - down(N) node N is refreshed by the server
- Corollary: there may be many *possible states* at each stage in a test; a test fails when there are no possible states that explain the observations that have been made

Modelling the state

- **Stable value** (i.e., the one on the server)
- **Conflict set** (only ever grows)
- For each node:
 - Current **local value**
 - "Fresh" or "stale"
 - "Clean" or "dirty"

i.e., has the global value changed since this node's last communication with the server

i.e., has the local value been written since this node was last refreshed by the server

Modelling the operations

- $\text{read}(N) \rightarrow V$
 - *Valid when:* $V =$ local value on node N
 - *Action:* state unchanged
- $\text{write}(N, V_{\text{new}}) \rightarrow \text{Vold}$
 - *Valid when:* $\text{Vold} =$ local value on node N
 - *Action:* local value on node N becomes V_{new}
node N becomes dirty
- $\text{stabilize}() \rightarrow (V, C)$
 - *Valid when:* $V =$ global value, $C =$ global conflict set, all nodes fresh and clean
 - *Action:* state unchanged

Modelling the operations continued...

- `down(N)`

- *Valid when*: N is neither dirty nor fresh
- *Action*: take N's local value from global value
N becomes fresh

Modelling the operations...

- up(N)

- *Valid when*: node N is dirty

- *Action*: **if** node N is fresh **then**

- take global value from node N's local value

- N becomes clean

- other nodes become stale*

- else**

- add N's local value to conflicts

- N becomes clean

Surprise!

write(N1, "a") → missing
write(N2, "a") → missing
stabilize() → ("a", {})

previous value was
"missing", i.e., N2
has not seen the
write of "a" yet

stable value

conflict set
(empty!!)

Specification and implementation disagree...
Is it a feature or a bug?

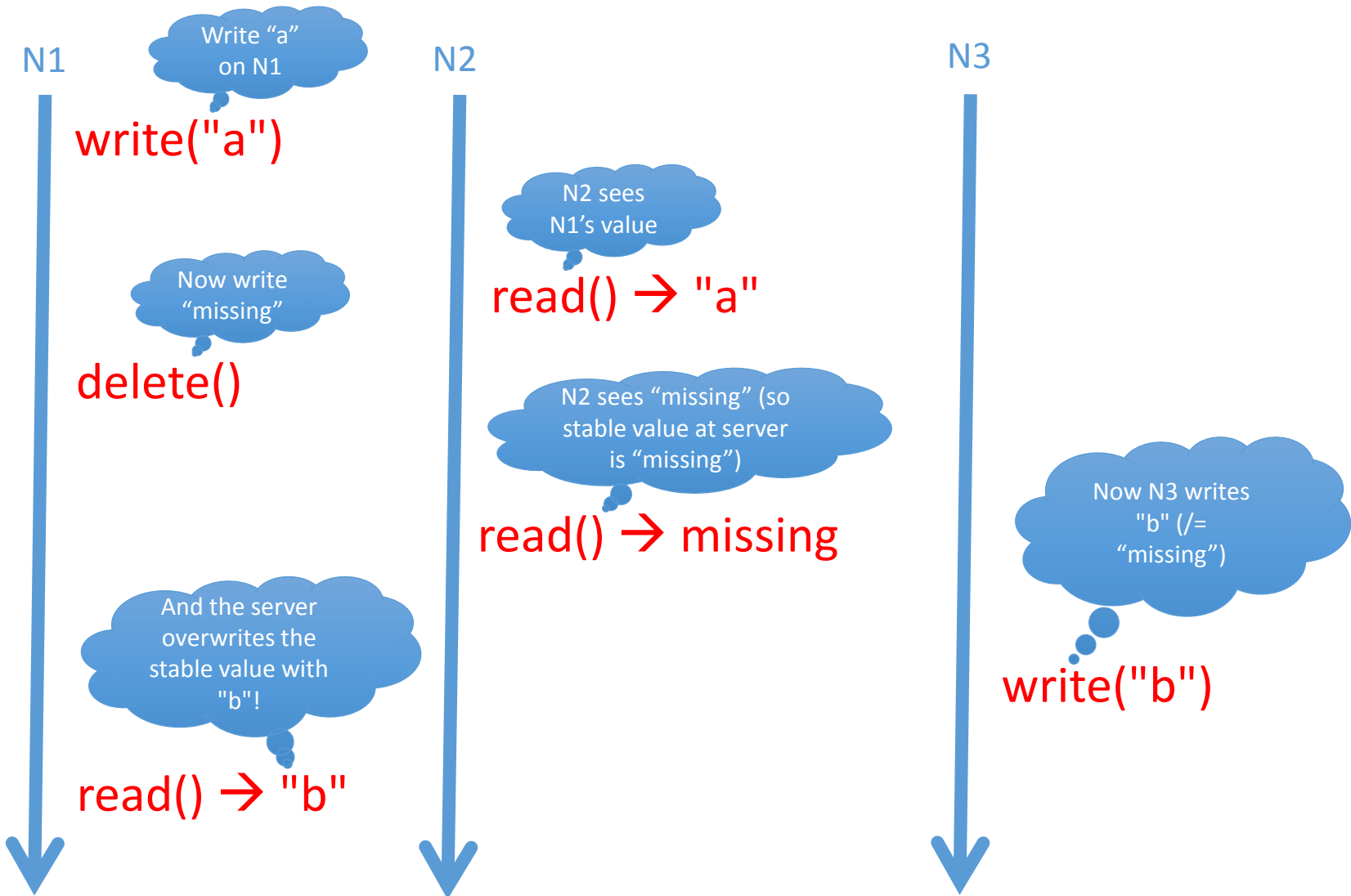
Refining the specification...

- Add special cases in specifications of up and down commands when the local and global values are identical
- The test now passes

Dealing with Deletion

- Deletion can easily be added to the model:
`delete(N)` just means `write(N, missing)`
- Try adding this and run some tests...

Surprise!

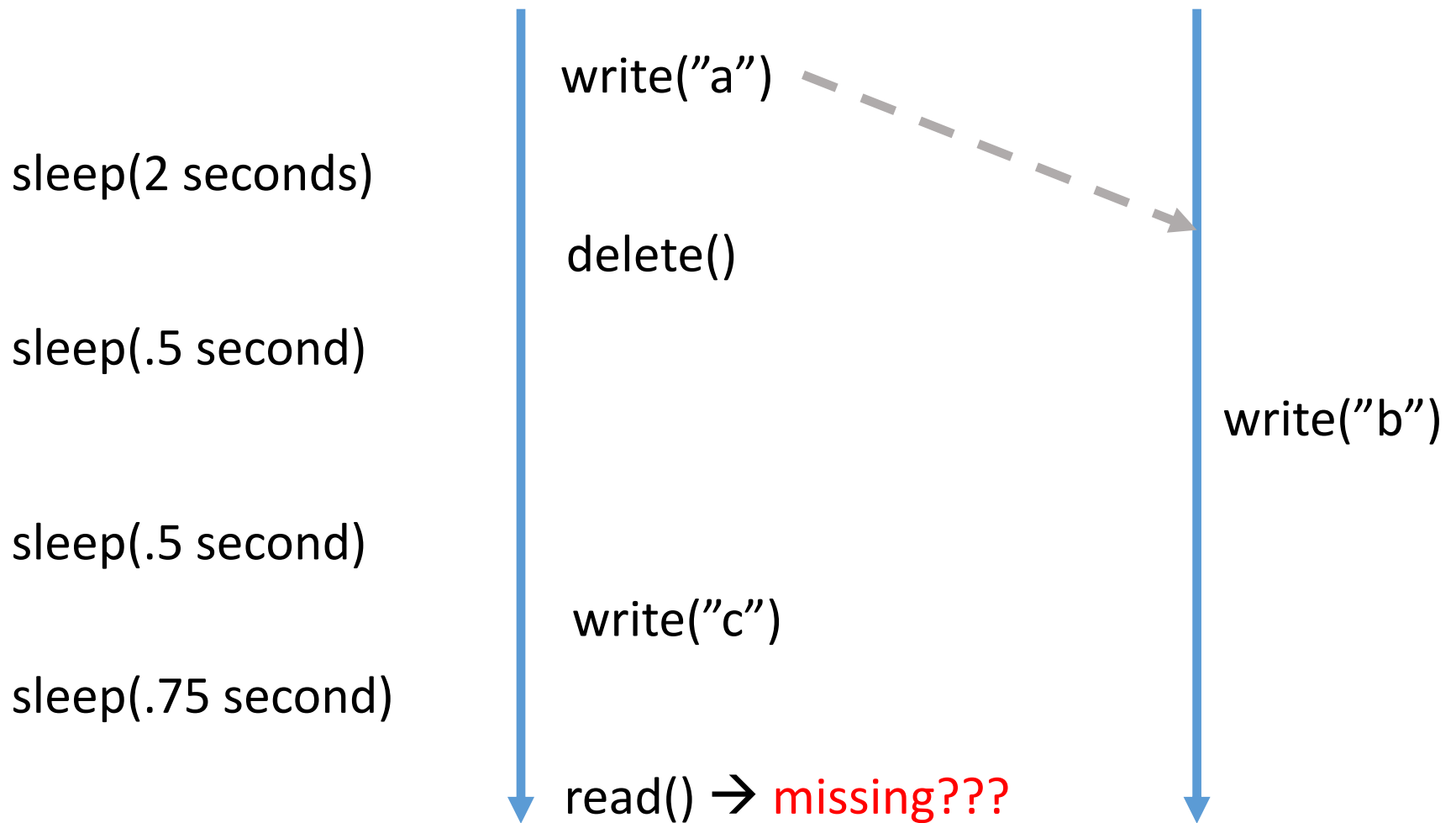


Refining the specification...

- Add special cases for “missing” in up and down actions:
 - When “missing” encounters another value during an up or down, the other value always wins
 - I.e., when a write and a delete conflict, the delete gets undone

And try some more tests...

Another surprise!



An Even Bigger Surprise!

```
write(N1, "b") -> missing  
sleep(.75 second)  
delete(N1) -> "b"  
sleep(.75 second)  
read(N1) -> "b"
```



b came back after being deleted!?!

Work in progress!

- Current state:
 - Formal specification of single-file behavior of Dropbox and related services
 - One apparent bug in Dropbox so far
 - Prototype validation harness in Erlang QuickCheck
- Next steps:
 - Add directories
 - Test *your* synchronizer :-)

Resources

- Unison
www.cis.upenn.edu/~bcpierce/unison/
- Unison specification
google “What is a File Synchronizer?”
google “What’s in Unison?”
- Quviq testing tools
www.quviq.com
- Lenses
<http://cis.upenn.edu/~bcpierce/papers/index.shtml#Lenses>
(a more general theory of bidirectional information propagation between related structures)

The logo for Unison, featuring the word "Unison" in a stylized, bold font. The letters are filled with a gradient from yellow to orange, and the 'U' has a yellow arrow pointing upwards.The logo for QuviQ, featuring the word "QuviQ" in a bold, black, sans-serif font. Below the 'i' and 'v' are three small orange dots.