

The Mysteries of Dropbox

John Hughes



QuviQ

File Synchronizer Usage



400 million (June 2015)



240 million (Oct 2014)

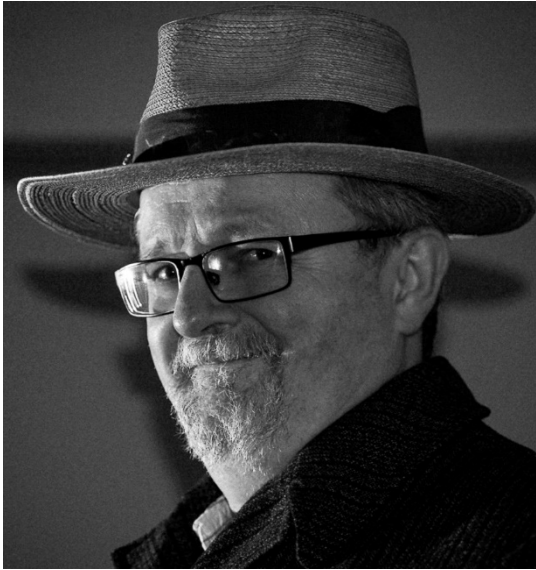


250 million (Nov 2014)

Are they trustworthy?

What do they do?

How can we test them?



**TEST
ING**



Hand written test cases



Generated test cases

QuickCheck



1999—invented by Koen Claessen and myself, for Haskell

2006—Quviq founded marketing Erlang version

Many extensions

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

Why Generate Tests?

- Much wider variety!
- More confidence!
- Less work!

Example: Testing a Queue with QuickCheck

- API:
 - `q:new(Size)` – create a queue
 - `q:put(Q, N)` – put N into the queue
 - `q:get(Q)` – remove and return the first element

A Generated Failing Test

```
q:new(1) -> {ptr,...}  
q:put({ptr,...}, 1) -> ok  
q:get({ptr,...}) -> 1  
q:put({ptr,...}, -1) -> ok  
q:get({ptr,...}) -> -1  
q:put({ptr,...}, 0) -> ok  
q:put({ptr,...}, 1) -> ok  
q:put({ptr,...}, 0) -> ok  
q:put({ptr,...}, -1) -> ok  
q:get({ptr,...}) -> -1
```

**Quite
long and
boring!**

Reason:

Post-condition failed:

-1 /= 0

A *Shrunk* Failing Test

We made a queue of size 1...

```
q:new(1) -> {ptr, ...}  
q:put({ptr, ...}, 0) -> ok  
q:put({ptr, ...}, 1) -> ok  
q:get({ptr, ...}) -> 1
```

...and put TWO things into it!

Reason:

Post-condition failed

1 /= 0



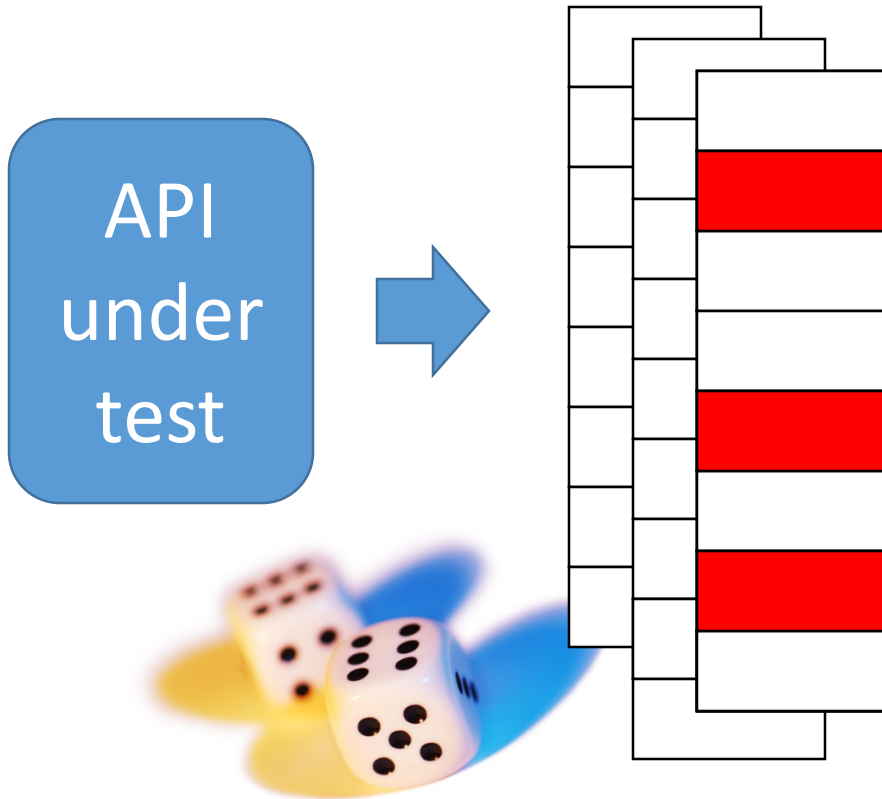
Bug in the code

We should have got
an exception

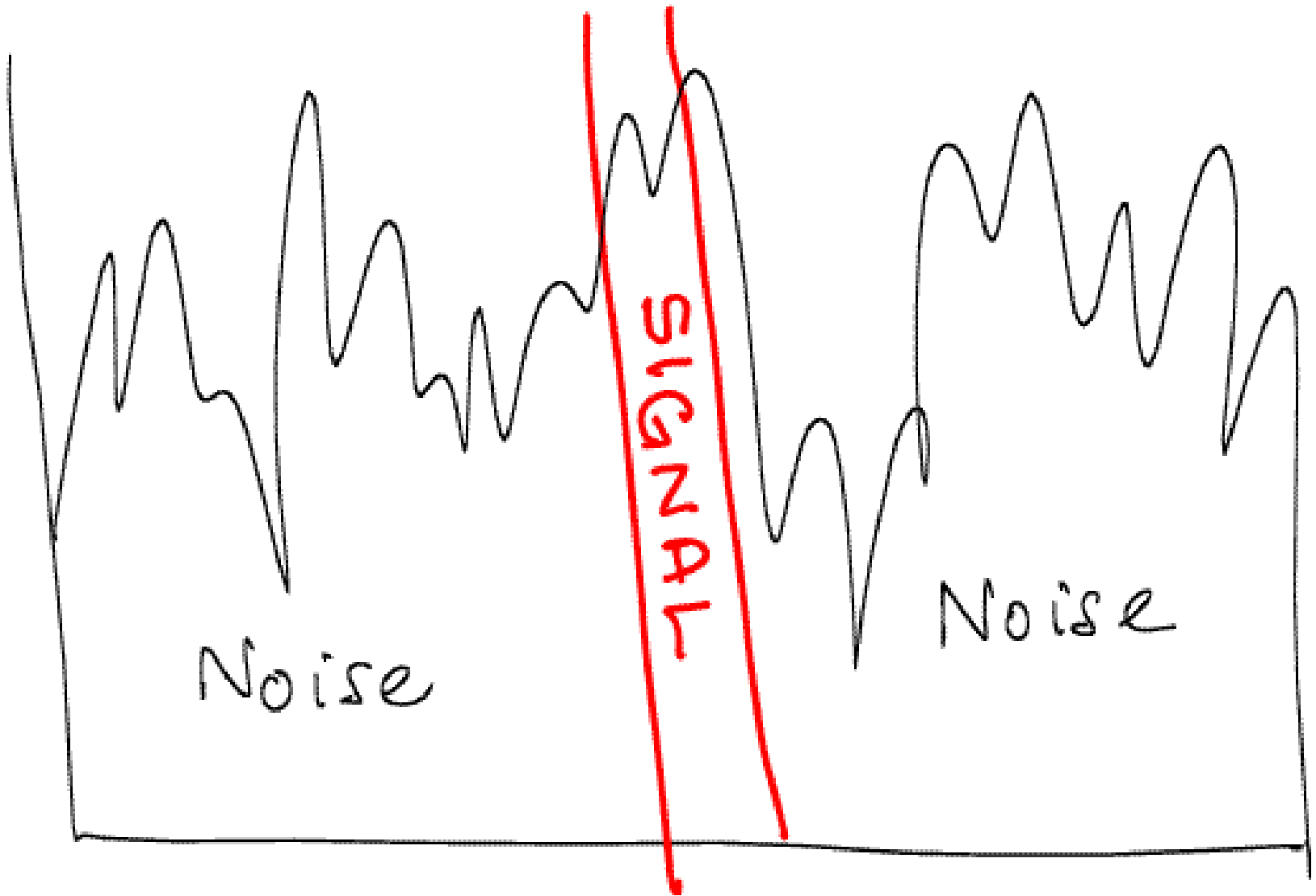
Bug in the test

We shouldn't
generate nonsense
tests that abuse the
API

QuickCheck



A minimal failing
example



arbitrary

How can we tell if a test passed?







State
transitions



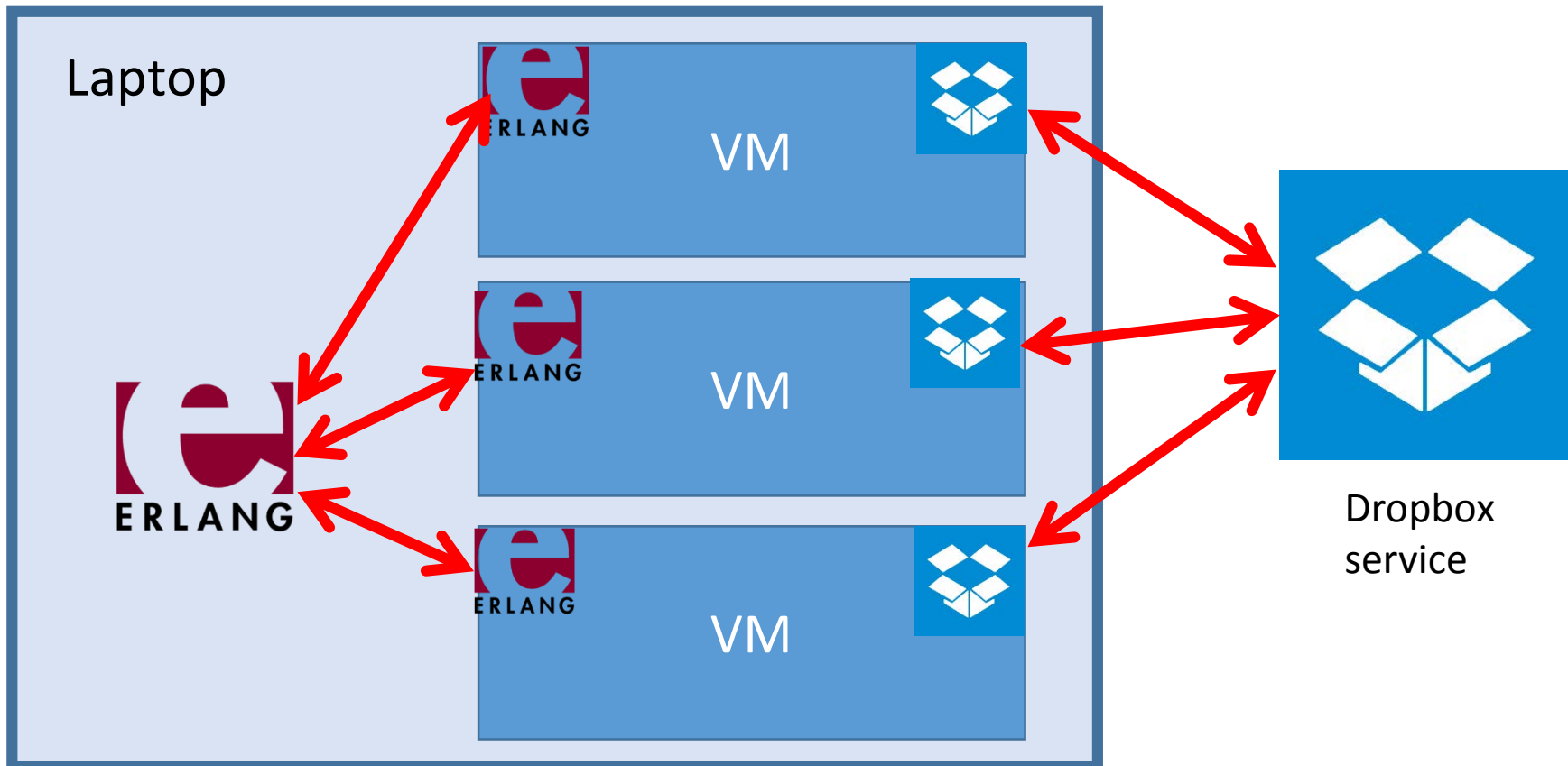
Postconditions

Modelling a queue

		Model	Postcondition
	<code>new(1)</code>	<code>[]</code>	<code>result/=NULL</code>
	<code>put(...,0)</code>	<code>[0]</code>	
	<code>put(...,1)</code>	<code>[0,1]</code>	
	<code>get(...)</code>	<code>[1]</code>	<code>result==0</code>

But what about Dropbox?

Read and
write files

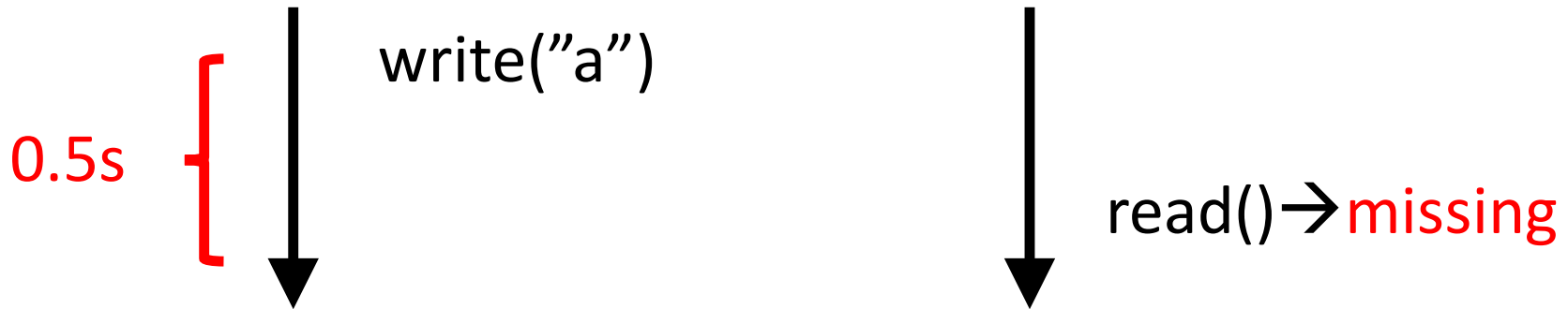


Goals

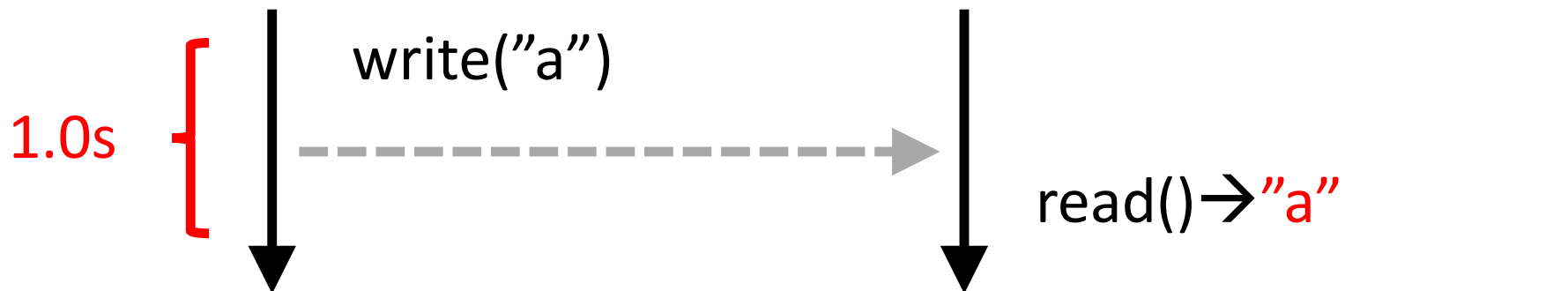
- A simple model of what a file synchronizer does, that the *user* can understand without reference to implementation details
- Ideally, a model that works for many different synchronizers

What's the model?

- Contents of each file?

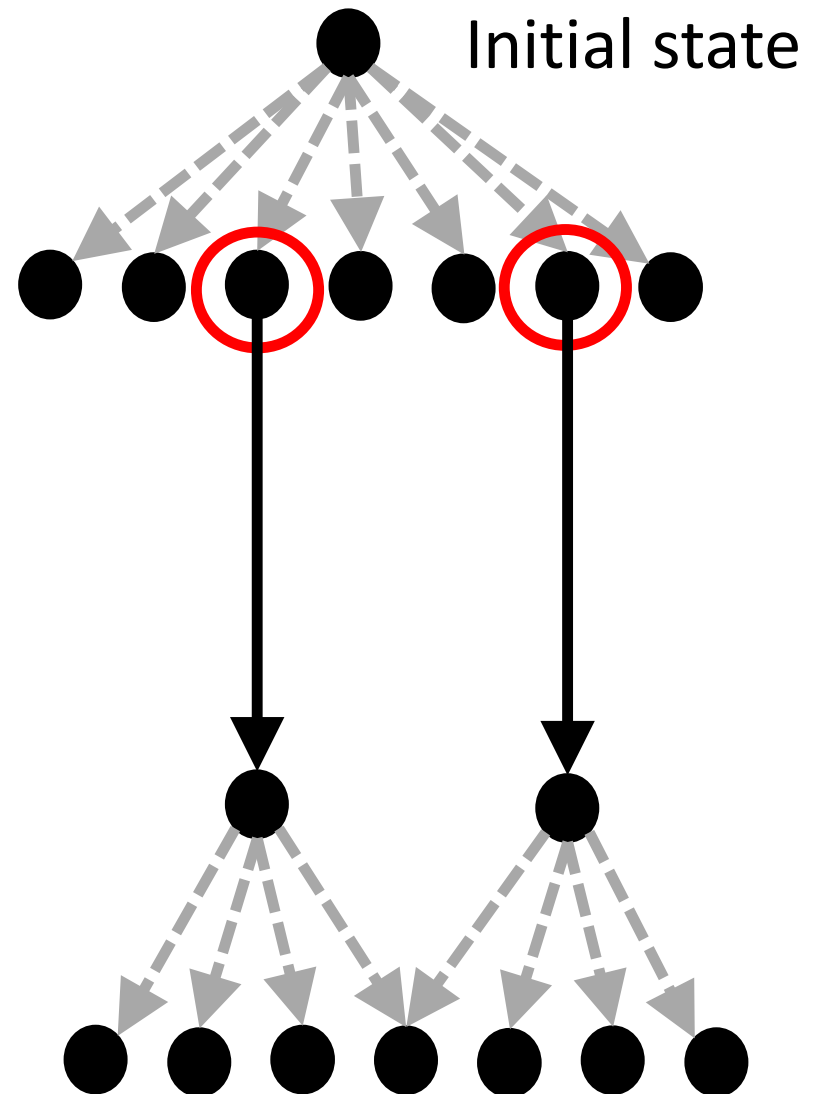
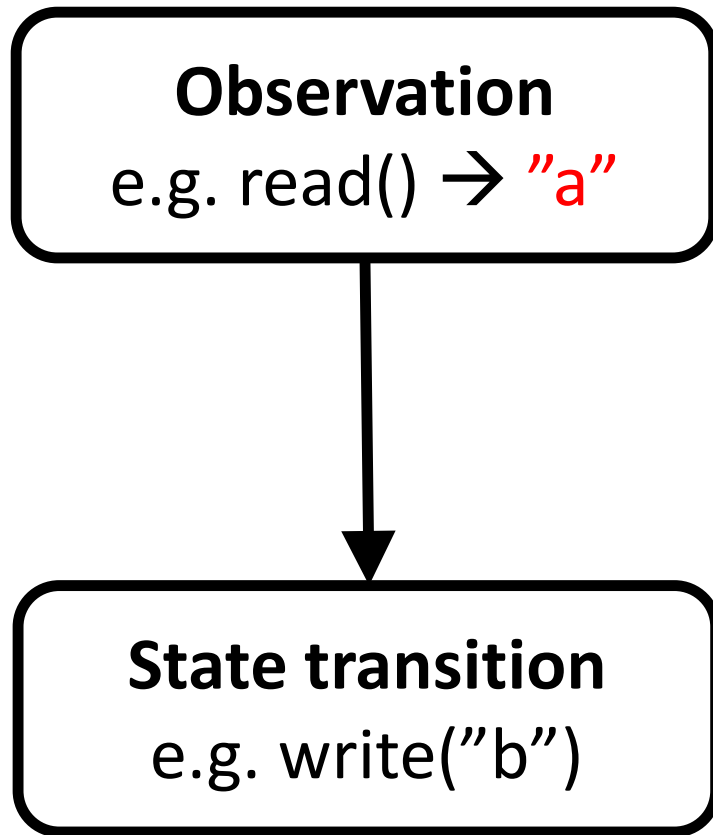


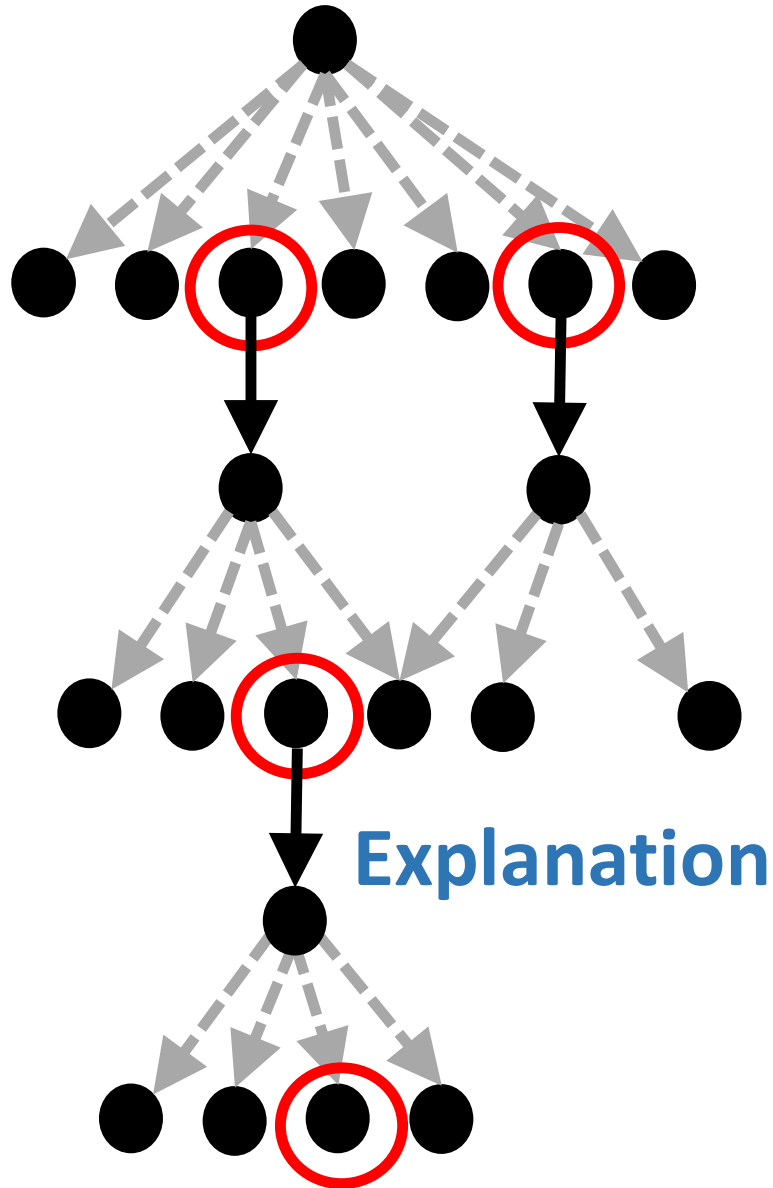
- Contents of each file on each node?



All possible background actions

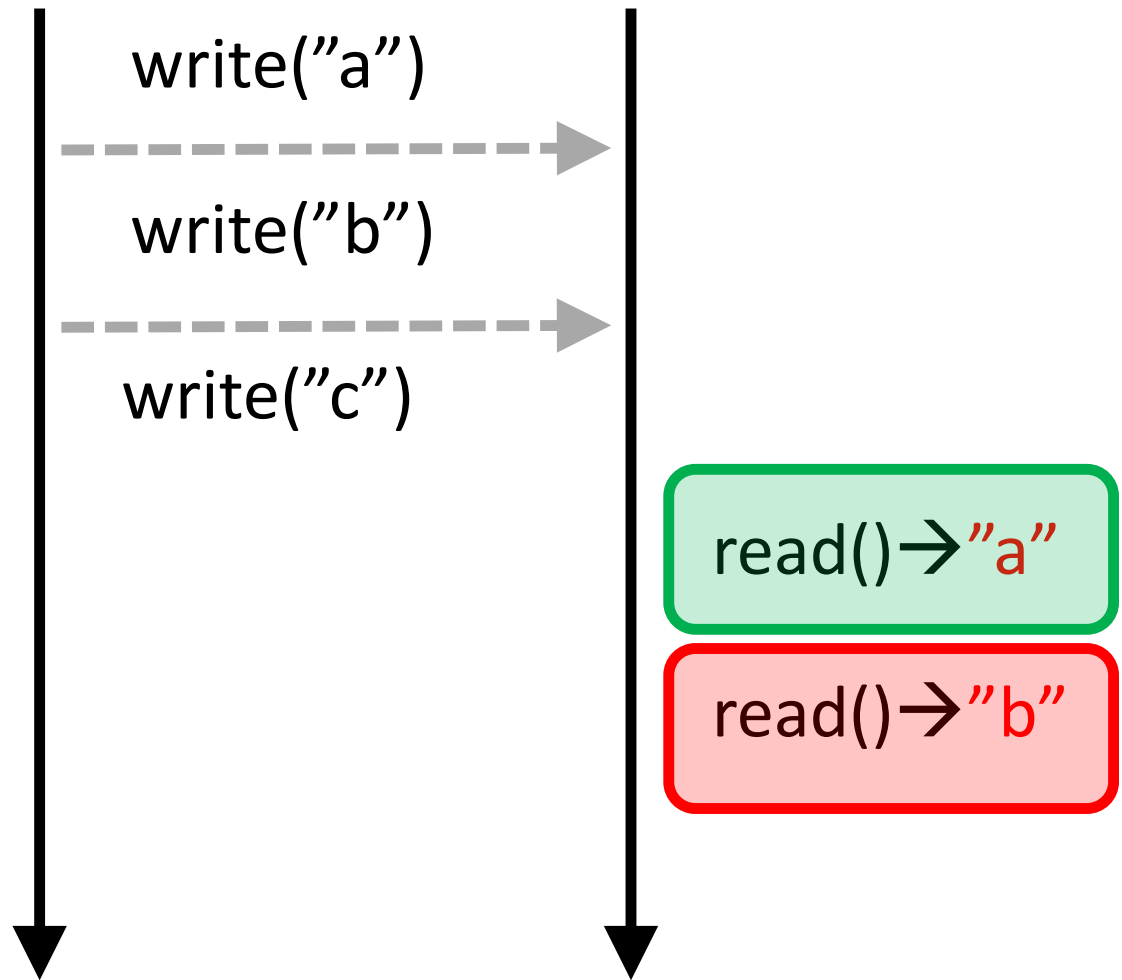
A new approach



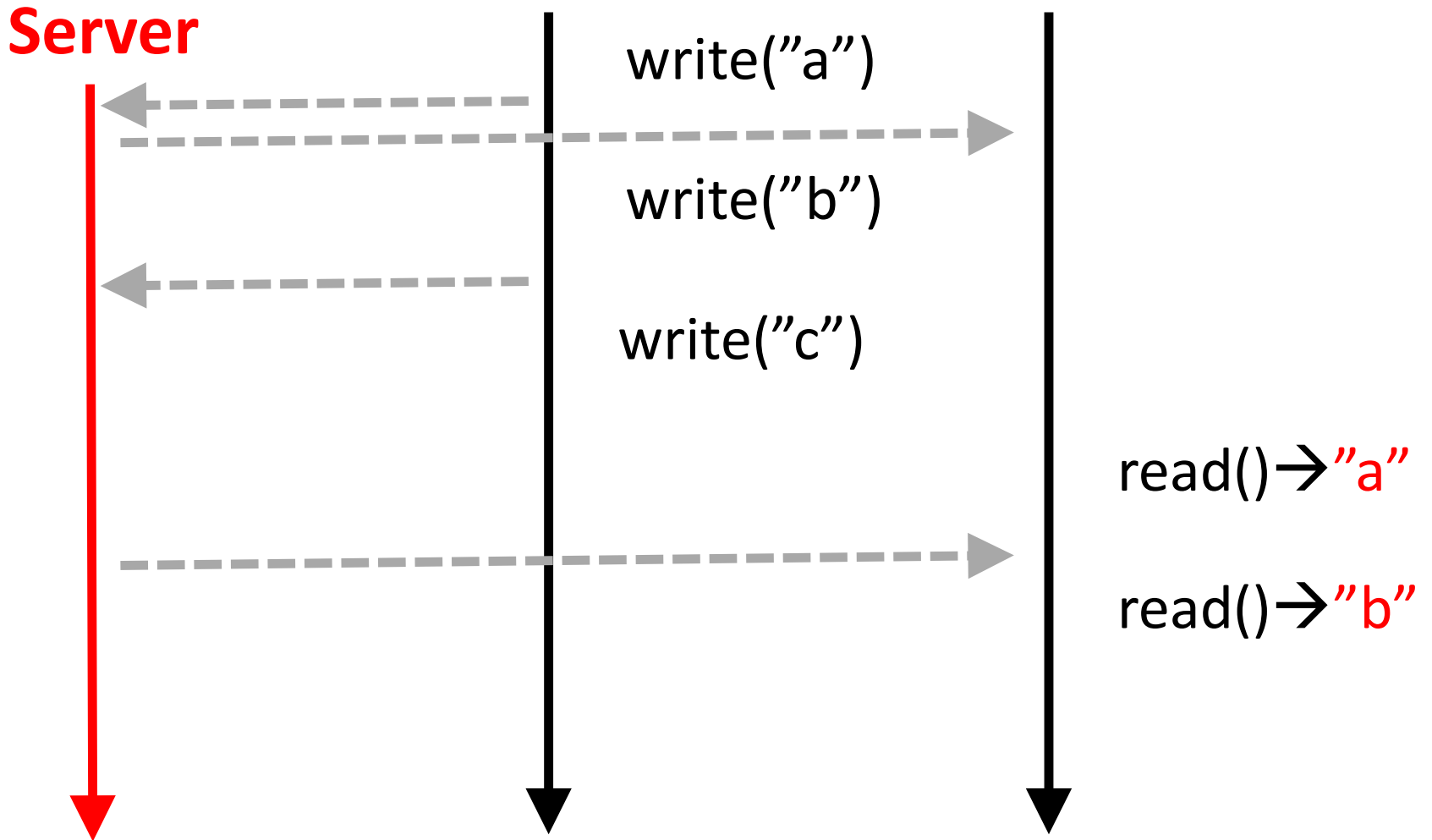


No matching observations means the test fails

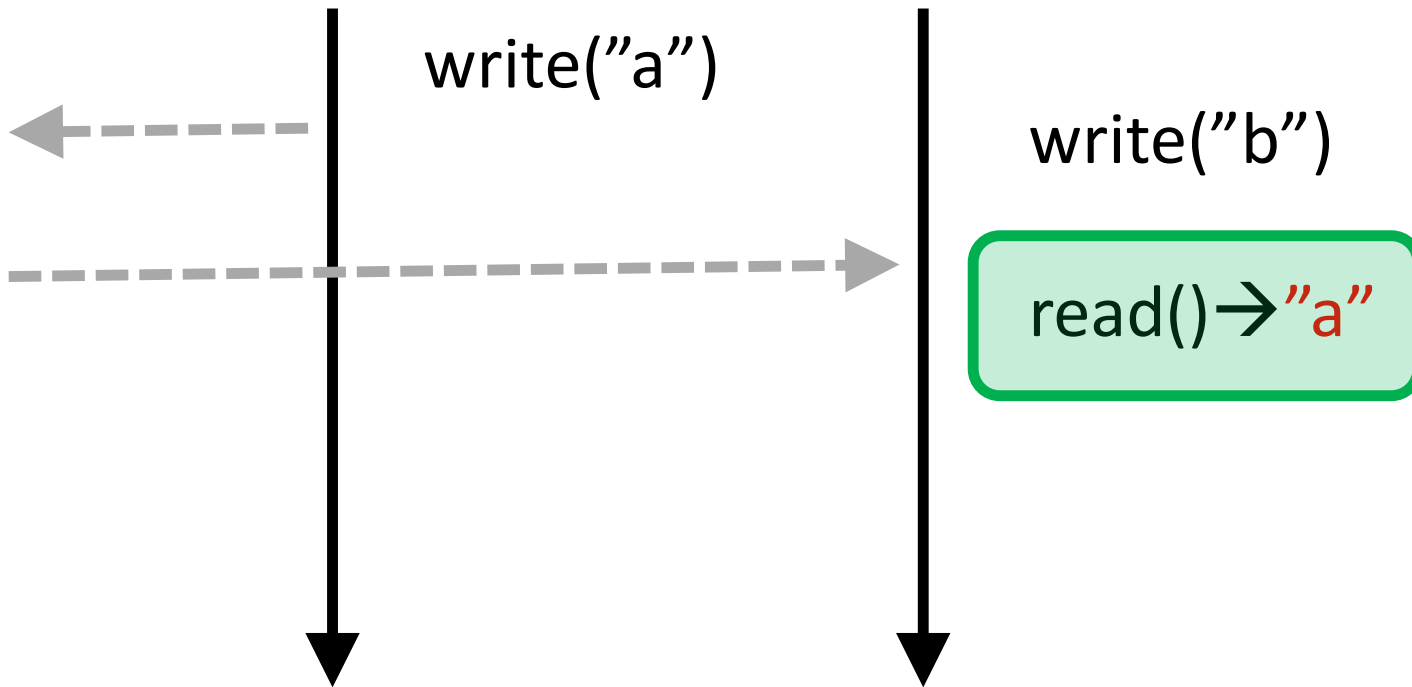
What background operations?



What background operations?



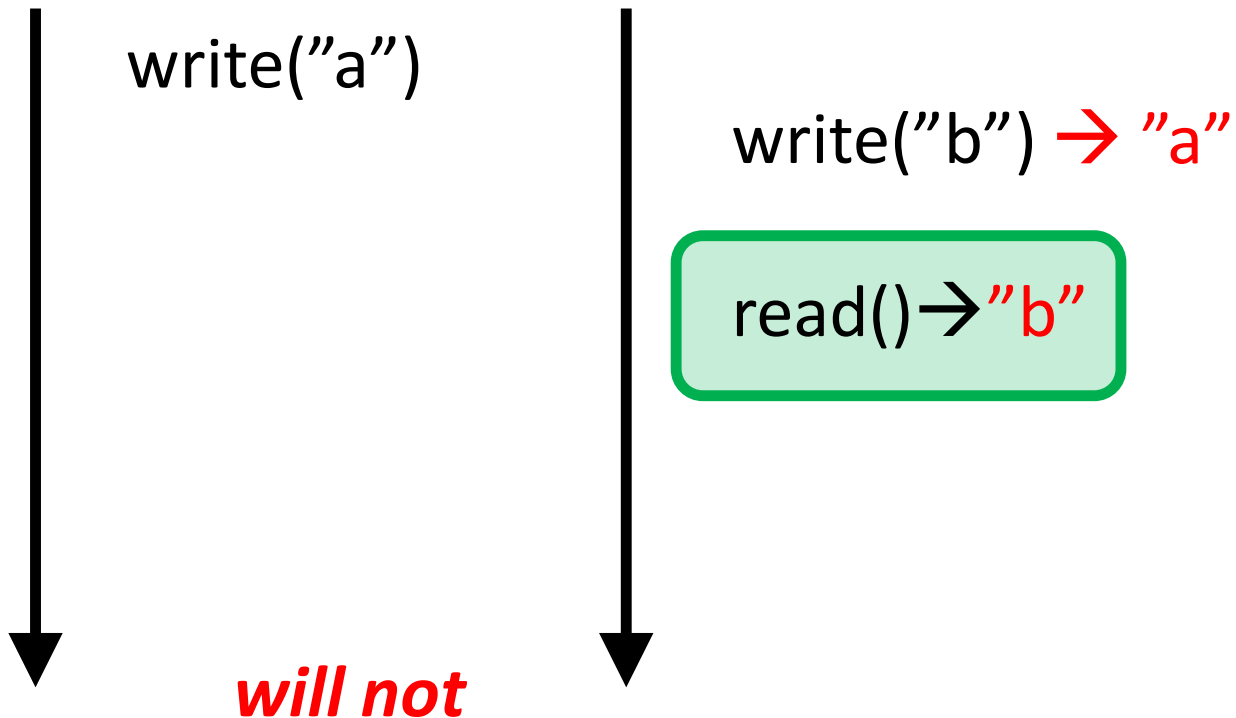
Conflicts



`"b" will appear in a conflict file`

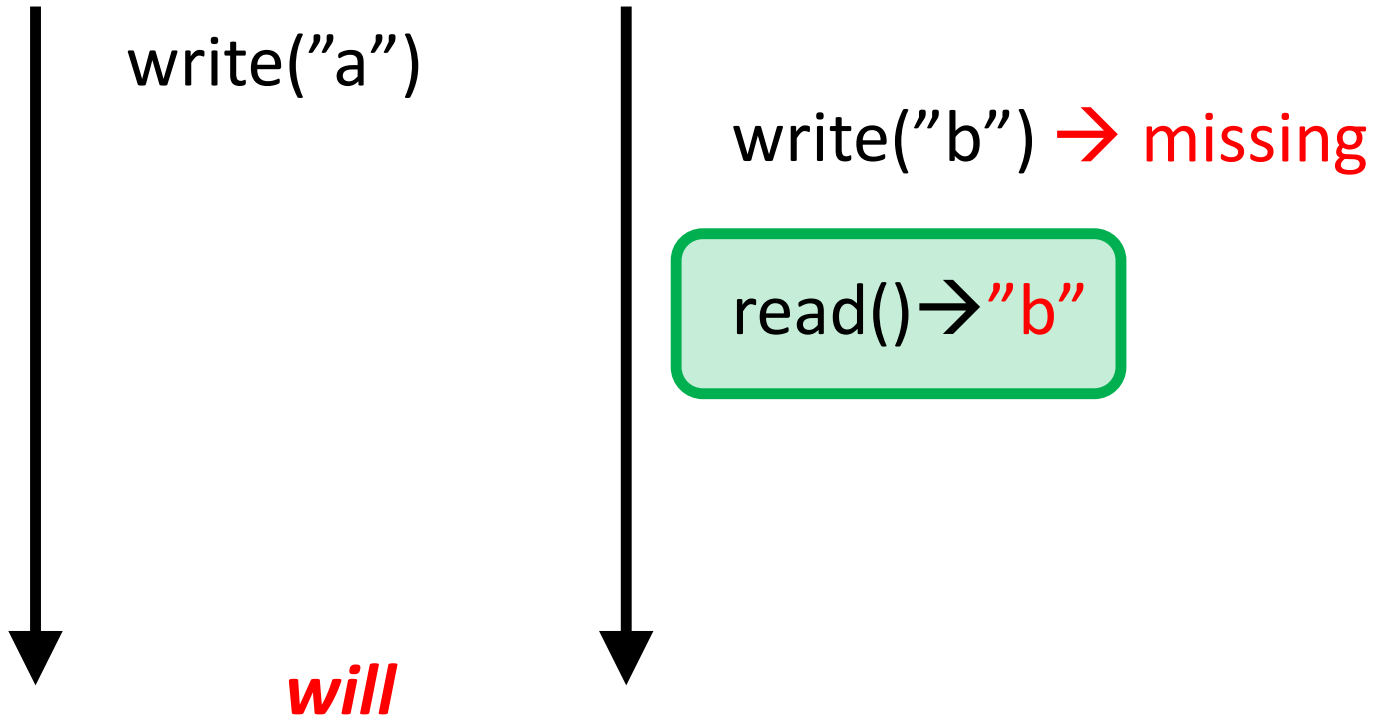
Conflicts

Observe the *value overwritten*



"a" may or may not appear in a conflict file

Conflicts



"a" may or may not appear in a conflict file

Observing conflict files

- Conflict files *do not appear immediately!*
 - Wait for a stable state to check for them

stabilize() → (V,C)

*The final value in the file
(that all nodes converge to)*

*The set of values in
conflict files*

stabilize() → (V,C)

How long should we wait?
BUT NOT

- Until all nodes agree on file contents (V) and conflict files (C)

TOO

- Until the Dropbox daemon on each node claims to be idle

LONG!!!

- **Until we see what we expect!**

Our model

- Global value
- Global conflict set



On the "server"

- For each node:
 - Local value
 - "Fresh" or "Stale"
 - "Clean" or "Dirty"

"Stale" means
needs to download
from the server

"Dirty" means
needs to upload to
the server

read() → v

Observes:

Local value is V

State transition:

None

`write(Vnew) → Vold`

Observes:

Local value is Vold

State transition:

Local value becomes Vnew

This node becomes dirty

stabilize() \rightarrow (V,C)

Observes:

Global value is V

Conflict set is C

All nodes are fresh and clean

State transition:

None

download()

Observes:

This node is stale and clean

State transition:

Local value becomes global value

This node becomes fresh

upload()

Observes:

This node is dirty

State transition:

First upload wins

This node becomes clean

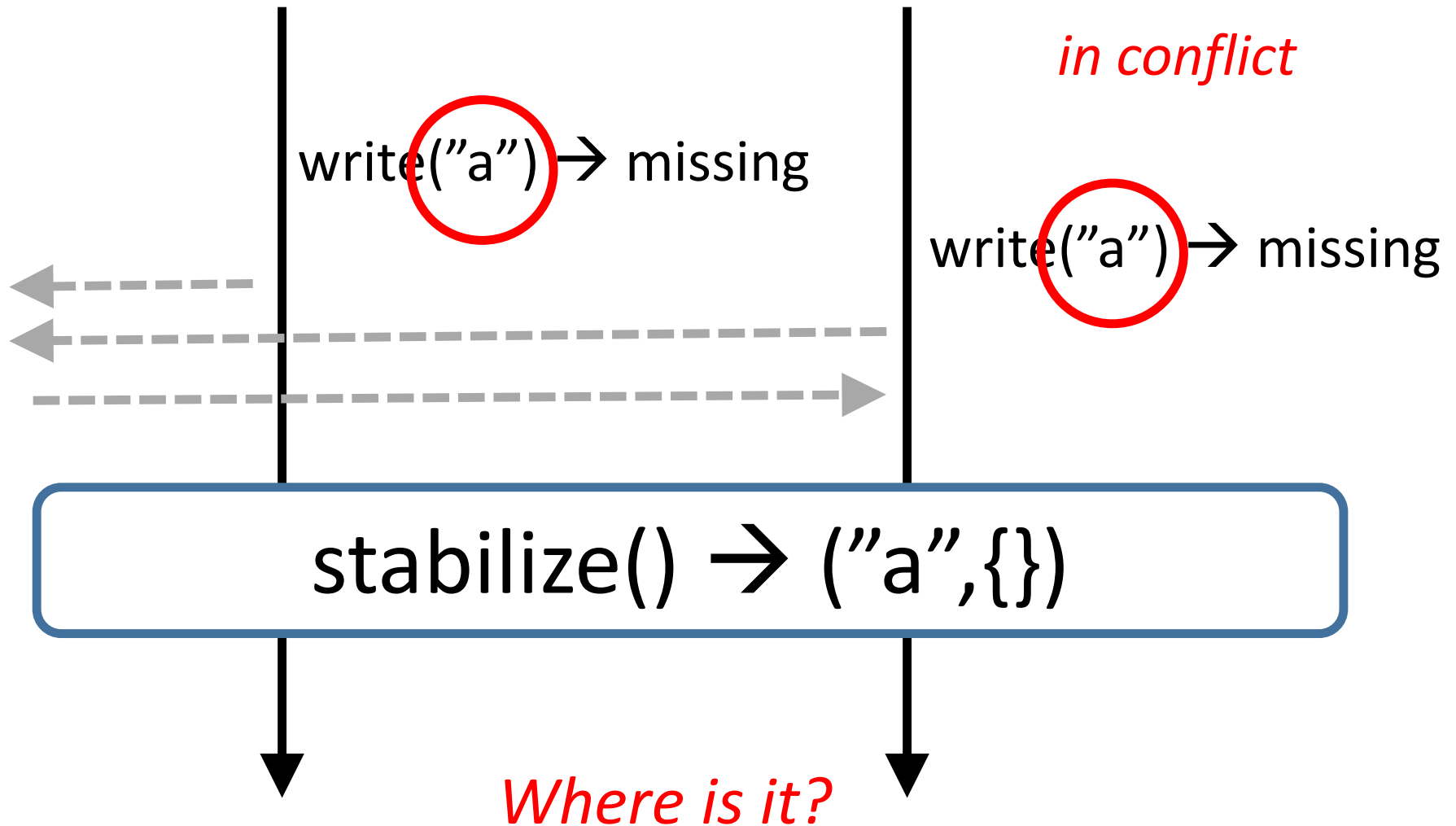
if this node is fresh

then Global value becomes local value

All other nodes become stale

else Local value is added to conflicts

Does the model match reality?



A value does not
conflict with itself

upload()

Observes:

This node is dirty

State transition:

This node becomes clean

if local value \neq global value then

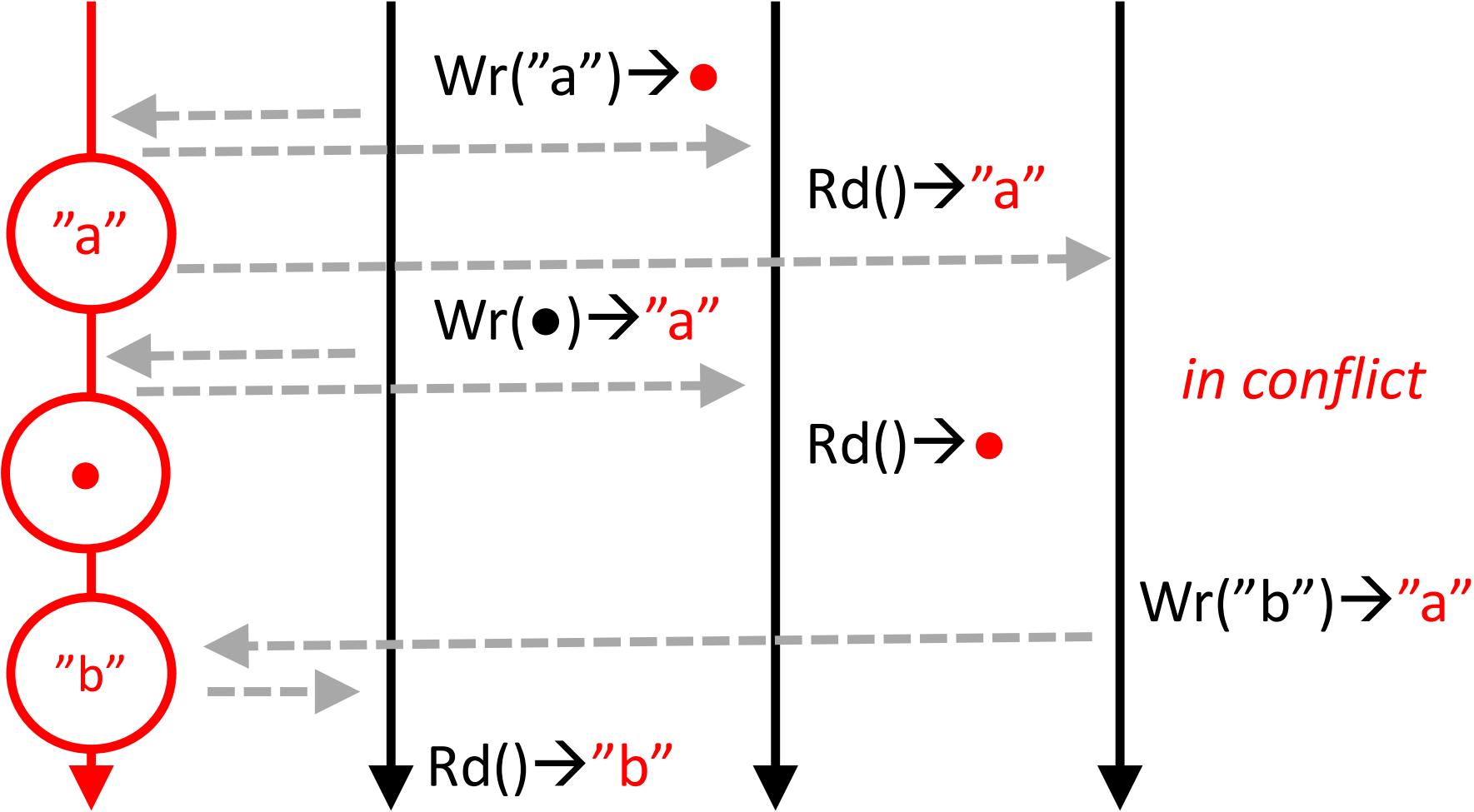
if this node is fresh

then Global value becomes local value

All other nodes become stale

else Local value is added to conflicts

Another inconsistency



But "b" should be in a conflict file!

'missing' loses every
conflict

upload()

Observes:

This node is dirty

State transition:

This node becomes clean

if local value \neq global value **then**

if this node is fresh **or** global value is missing

then Global value becomes local value

All other nodes become stale

else Local value is added to conflicts

So far...

- *We're fitting the model to the implementation*

Why?

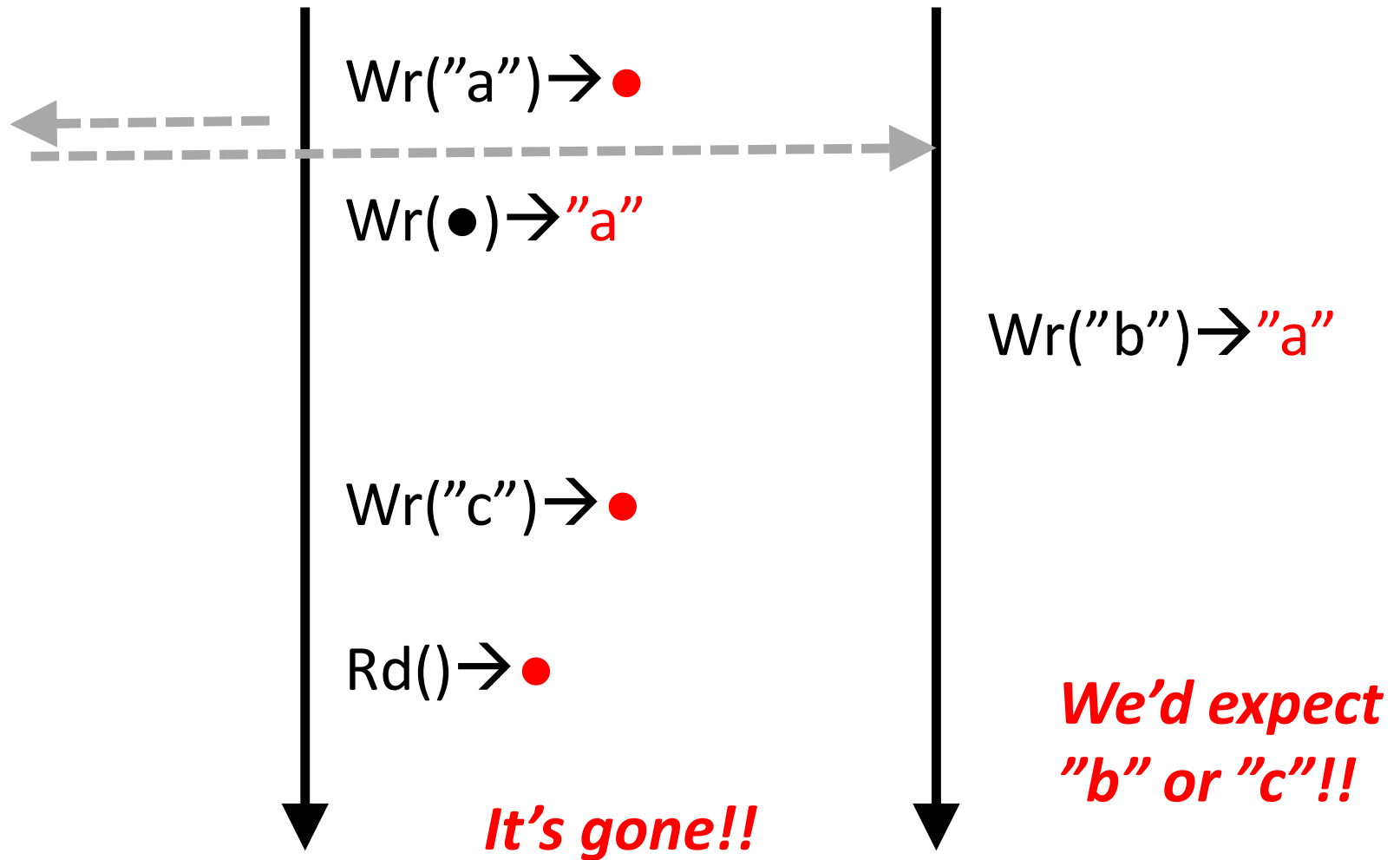
- Because Dropbox have thought harder about synchronization than we have!

For each inconsistency:

- Ask "*Is this the intended behaviour?*"

Surprises

Dropbox can delete a newly created file



Dropbox can recreate deleted files

Wr("a") → ●

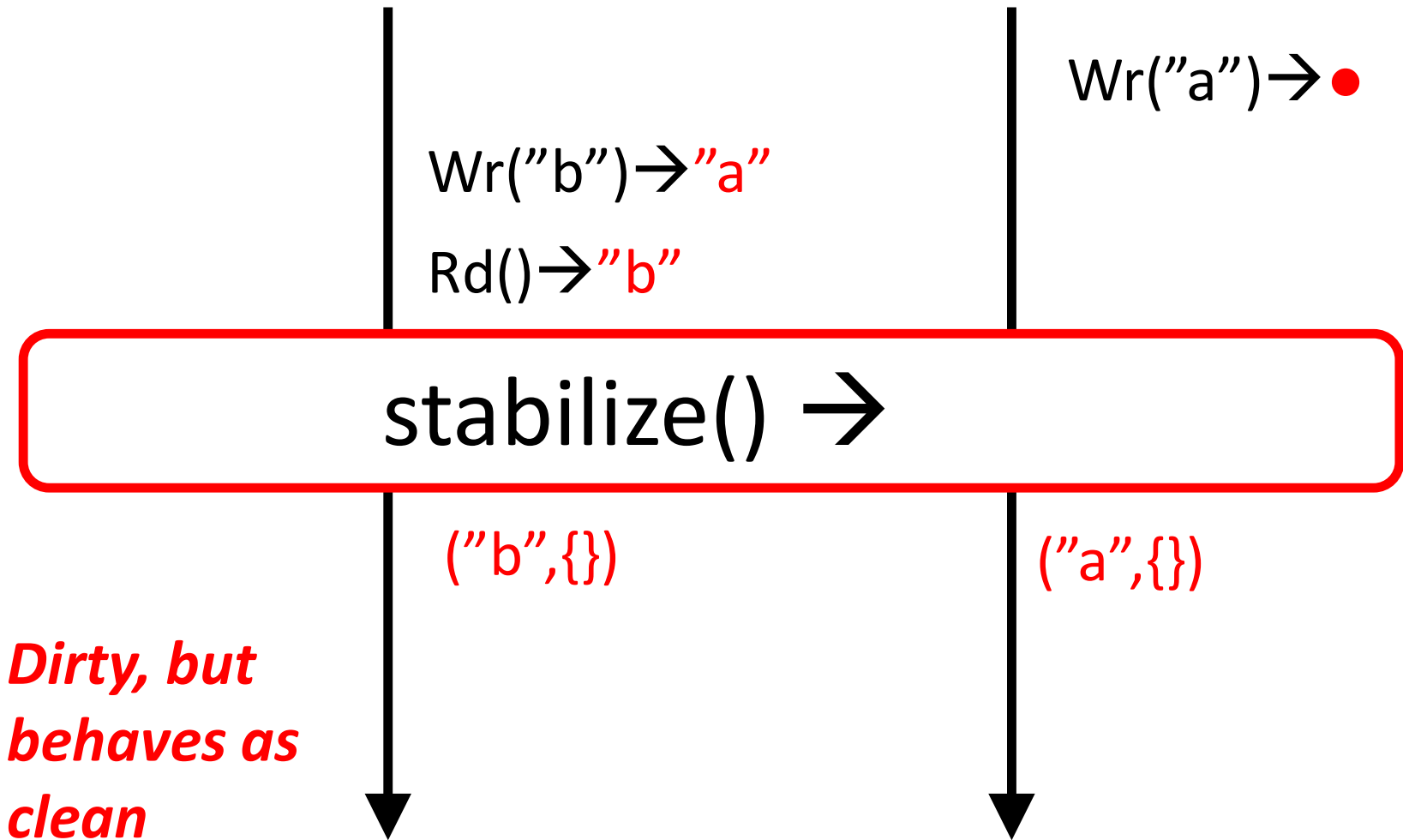
Wr(●) → "a"

Rd() → "a"

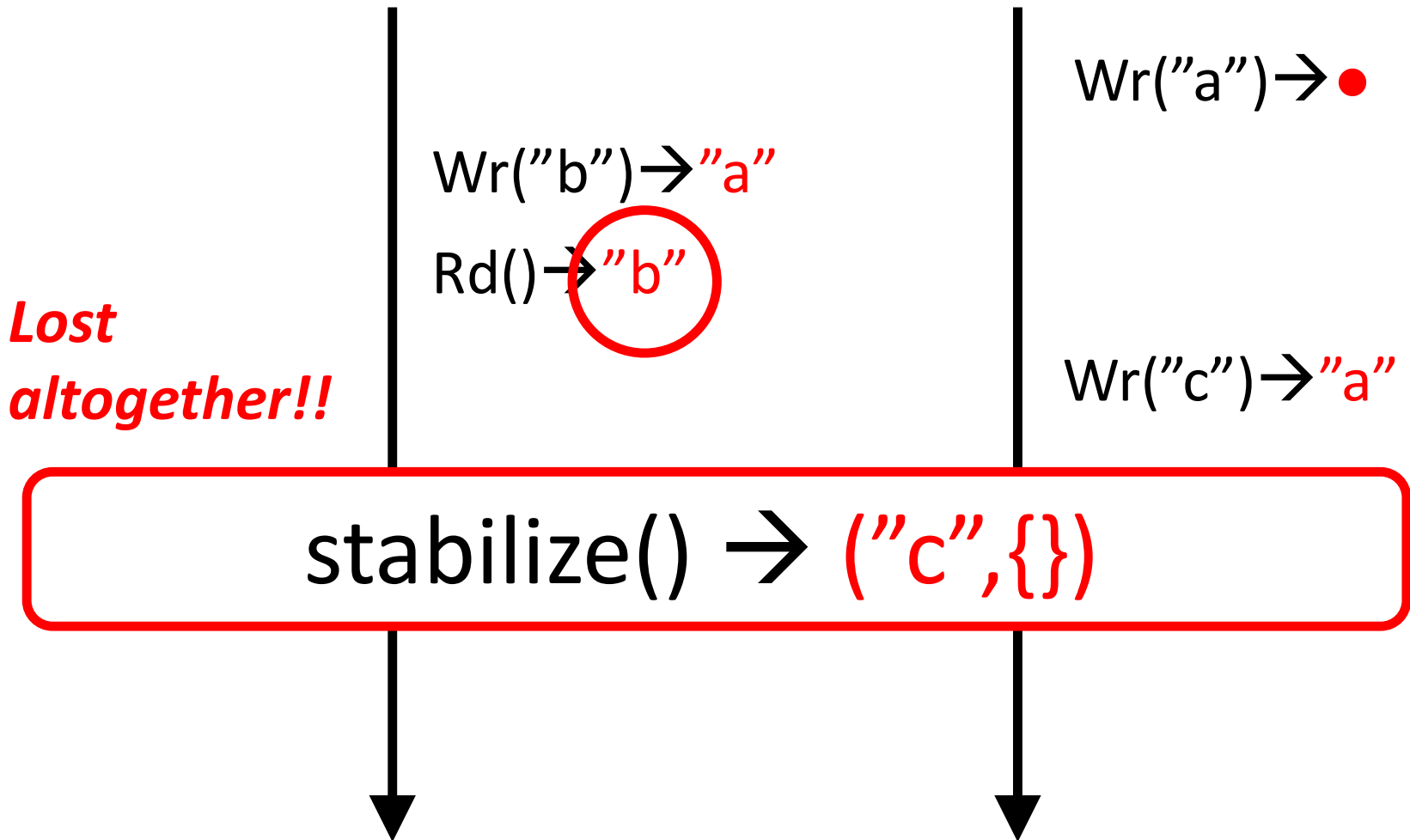
What??

stabilize() → ("a", {})

Dropbox can lose data completely



Dropbox can lose data completely



What did we do?

- Tested a non-deterministic system by *searching for explanations* using a model with hidden actions
- Used QuickCheck's minimal failing tests to *refine* the model, until it matched the intended behaviour
- Now minimal failing tests reveal *unintended* system behaviour

What do Dropbox say?

- The synchronization team has reproduced the buggy behaviours
- They're *rare* failures which occur under very special circumstances
- They're developing fixes

Synchronization is subtle!

- There's much more to do...
- Add directories!
 - Directories and files with the same names
 - Conflicts between deleting a directory and writing a file in it
 - ...
- More file synchronizers!



Mysteries of Dropbox

Property-Based Testing of a Distributed Synchronization Service

John Hughes*[†], Benjamin C. Pierce[‡], Thomas Arts*, Ulf Norell*[†],

* Quviq AB, Göteborg, Sweden

[†] Dept of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden

[‡] Dept of Computer and Information Science, University of Pennsylvania, PA, USA

Abstract—File synchronization services such as Dropbox are used by hundreds of millions of people to replicate vital data. Yet rigorous models of their behavior are lacking. We present the first formal—and testable—model of the core behavior of a modern file synchronizer, and we use it to discover surprising behavior in two widely deployed synchronizers. Our model is based on a technique for testing nondeterministic systems that avoids requiring that the system’s internal choices be made visible to the testing framework.

I. INTRODUCTION

File synchronization services—distributed systems that maintain consistency among multiple copies of a file or directory structure—are now ubiquitous. Dropbox claim 400 million users,¹ while Google Drive and Microsoft OneDrive are reported to have over 240 million users each.² In addition to these large-scale commercial offerings and many smaller ones, there are a plethora of open source synchronizers, enabling users to create their own ‘cloud storage.’ With so many people trusting their data to synchronization services, their correctness should be a high priority indeed.

Surprisingly, then, it seems that only one file synchronizer has been formally specified to date: Unison [11]–[13]. However,

Our goal in this paper is to present a testable formal specification for the core behavior of a file synchronizer. We do so via a *model* developed using Quviq QuickCheck [4]. Despite the apparent simplicity of the problem, we encountered interesting challenges regarding both specification and testing. We used the model to test Dropbox, Google Drive, and ownCloud (an open source alternative), exposing unexpected behavior in two out of three.

In Section II, we introduce our testing framework. Section III gives a high-level overview of the concepts used in our model, in particular the *operations* performed by test cases, the *observations* made when a test case is run on the system under test (SUT), and the *explanations* that we construct to determine whether the test has passed or failed. Section IV presents the formal specification itself, beginning with a naive version and refining it in light of failed tests that reveal subtleties in the synchronizer’s handling of corner cases. Section V describes further failed tests that, rather than pinpointing inadequacies in the specification, seem to us to exemplify unintended behaviors of both Dropbox and ownCloud, including situations where each system can lose data. In Section VI, we discuss the pragmatics of our testing framework, in particular our methods

Coming out in
April,

IEEE
International
Conference
on Software
Testing,
Chicago

www.quviq.com