Mysteries of Dropbox

Property-Based Testing of a Distributed Synchronization Service

John Hughes, Benjamin Pierce, Thomas Arts, Ulf Norell
Synchronization Services

- **Dropbox**: 400 million (June 2015)
- **Google Drive**: 240 million (Oct 2014)
- **OneDrive**: 250 million (Nov 2014)
Are they trustworthy?

(exactly!)

What do they do?

Can we test them?
Writing test cases by hand
(especially for testing distributed systems!)

Generate test cases from a model
Our Goals

• Develop a **precise specification** of the core behavior of a synchronization service
  • Phrased from the perspective of users
  • Applicable to a variety of different synchronizers

• Use **property-based random testing** to validate it against Dropbox’s observed behavior
Why Generate Tests?

• Much wider variety!
  • Crucial for effective testing of distributed services
    • Subtle edge cases, timing dependencies, ...

• More confidence!
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

API under test

A minimal failing example
Test = list of operations

System under test

Each operation gives rise to an observation

Op₁ → Obs₁
Op₂ → Obs₂
Op₃ → Obs₃

Model

Each observation induces a transition from one model state to the next

A test fails when we make an observation that is not allowed by the model
Test Harness for Dropbox

Client nodes

Controller process

Client 1

Client 2

Client 3

Dropbox server
What operations and observations do we need?
One Simplification…

• Real filesystem APIs are complex
  • Files, directories, timestamps, permissions, extended attributes, symlinks, hard links, …

• We make a small restriction…

Filesystem = single file
<table>
<thead>
<tr>
<th>Operations</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textsc{read}_n</td>
<td>\textsc{read}_n \rightarrow &quot;current value&quot;</td>
</tr>
<tr>
<td>\textsc{write}_n (&quot;new value&quot;)</td>
<td>\textsc{write}_n (&quot;new value&quot;) \rightarrow &quot;old value&quot;</td>
</tr>
</tbody>
</table>

Use special value $\perp$ for “no file”

\textsc{read}_n \rightarrow \perp means that the file is missing
\textsc{write}_n (\perp) means delete the file
Challenge #1: Conflicts

write("a")
write("b")

Dropbox’s answer:
The first value to reach the server wins; other values are moved to conflict files in the same directory.

However, these conflict files may not appear for a little while!
Second try…

<table>
<thead>
<tr>
<th>Operations</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{READ}</td>
<td>\text{READ} \rightarrow \text{“current value”}</td>
</tr>
<tr>
<td>\text{WRITE} (“new value”)</td>
<td>\text{WRITE} (“new value”) \rightarrow \text{“old value”}</td>
</tr>
<tr>
<td>\text{STABILIZE}</td>
<td>\text{STABILIZE} \rightarrow (“value”, {“conflict values”})</td>
</tr>
</tbody>
</table>

Same value in the file on all clients

Same set of values in conflict files on all clients
Challenge #2: Background operations

• The Dropbox client communicates with the test harness via the filesystem.

But…

• The Dropbox client *also* communicates with the Dropbox servers!
  • Timing of these communications is unpredictable

Invisible, unpredictable activity  ➔  Nondeterminism!
Approach

• Model the whole system state including the (invisible) state of the server

• Add "conjectured observations" to the ones we actually observe when running tests...
<table>
<thead>
<tr>
<th>Operations</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>READ</strong></td>
<td><strong>READ</strong> → “current value”</td>
</tr>
<tr>
<td><strong>WRITE</strong> (&quot;new value&quot;)</td>
<td><strong>WRITE</strong> (&quot;new value&quot;) → “old value”</td>
</tr>
<tr>
<td><strong>STABILIZE</strong></td>
<td><strong>STABILIZE</strong> → (&quot;value&quot;, {“conflict values”})</td>
</tr>
</tbody>
</table>

node $N$ uploads its value to the server
node $N$ is refreshed by the server
starting state
all possible sequences of Up/Downs
hypothetical states
real observation (invalid in most hypothetical states)

etc.

Explanation

No explanation
= failing test
Example:

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE <code>a</code></td>
<td>WRITE <code>b</code></td>
</tr>
<tr>
<td>READ</td>
<td>WRITE <code>c</code></td>
</tr>
<tr>
<td></td>
<td>STABILIZE</td>
</tr>
</tbody>
</table>

Test
### Test

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE ‘a’</td>
<td>WRITE ‘b’</td>
</tr>
<tr>
<td>READ</td>
<td>WRITE ‘c’</td>
</tr>
<tr>
<td></td>
<td>STABILIZE</td>
</tr>
</tbody>
</table>

### Observations

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE ‘a’ (\rightarrow \perp)</td>
<td>WRITE ‘b’ (\rightarrow ‘a’)</td>
</tr>
<tr>
<td>READ (\rightarrow ‘b’)</td>
<td>WRITE ‘c’ (\rightarrow ‘b’)</td>
</tr>
<tr>
<td>STABILIZE</td>
<td>(\rightarrow (‘c’, \emptyset))</td>
</tr>
</tbody>
</table>
Example:

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WRITE 'a'</strong></td>
<td><strong>WRITE 'b'</strong></td>
</tr>
<tr>
<td><strong>READ</strong></td>
<td><strong>WRITE 'c'</strong></td>
</tr>
<tr>
<td><strong>STABILIZE</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE 'a' $\rightarrow$ ⊥ UP</td>
<td></td>
</tr>
<tr>
<td>WRITE 'b' $\rightarrow$ 'a'</td>
<td></td>
</tr>
<tr>
<td>WRITE 'c' $\rightarrow$ 'b'</td>
<td></td>
</tr>
<tr>
<td>STABILIZE $\rightarrow$ ('c', ∅)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOWN</td>
</tr>
<tr>
<td>WRITE 'b' $\rightarrow$ 'a' UP</td>
</tr>
<tr>
<td>WRITE 'c' $\rightarrow$ 'b' UP</td>
</tr>
<tr>
<td>READ 'b'</td>
</tr>
<tr>
<td>DOWN</td>
</tr>
<tr>
<td>STABILIZE $\rightarrow$ ('c', ∅)</td>
</tr>
</tbody>
</table>
Using the model for testing

1. Generate a random sequence of operations $Op_1 \ldots Op_n$
2. Apply them to the system under test, yielding observations $Obs_1 \ldots Obs_n$
3. Calculate all ways of interleaving Up and Down observations with $Obs_1 \ldots Obs_n$
4. For each of these, check whether

   \[
   \text{init-state} \rightarrow \ldots \rightarrow Obs_1 \rightarrow \ldots \rightarrow Obs_2 \rightarrow \ldots \rightarrow \ldots \rightarrow Obs_n
   \]

   is a valid sequence of transitions of the model

4. If the answer is “no” for every possible interleaving, we have found a failing test; otherwise, repeat
Model states

• **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
Modeling the operations

**READ $\rightarrow V$**

*Precondition:* $\text{LocalVal}_N = V$
*Effect:* none

**WRITE $V_{\text{new}} \rightarrow V_{\text{old}}$**

*Precondition:* $\text{LocalVal}_N = V_{\text{old}}$
*Effect:* $\text{LocalVal}_N \leftarrow V_{\text{new}}$
$\text{Clean?}_N \leftarrow \text{DIRTY}$
Modeling the operations

**Stabilize** → (V, C)

**Precondition:**
- ServerVal = V
- Conflicts = C
- for all N, Fresh?_N = FRESH
- Clean?_N = CLEAN

**Effect:** none
Modeling the operations

**DOWN**

**Precondition:** $\text{Fresh?}_N = \text{STALE}$

$\text{Clean?}_N = \text{CLEAN}$

**Effect:** $\text{LocalVal}_N \leftarrow \text{ServerVal}$

$\text{Fresh?}_N \leftarrow \text{FRESH}$
Modeling the operations

\[
\text{UP}
\]

\textbf{Precondition: } \text{Clean}_N = \text{DIRTY}

\textbf{Effect: } \text{Clean}_N \leftarrow \text{CLEAN}

\text{if } \text{Fresh}_N = \text{FRESH} \text{ then}

\text{if } \text{LocalVal}_N \neq \text{ServerVal} \text{ then}

\text{Fresh}_{N'} \leftarrow \text{STALE} \text{ for all } N' \neq N

\text{ServerVal} \leftarrow \text{LocalVal}_N

\text{else}

\text{if } \text{LocalVal}_N \neq \text{ServerVal} \text{ then}

\text{Conflicts} \leftarrow \text{Conflicts} \cup \{\text{LocalVal}_N\}
Dealing with deletion

• Deletion can easily be added to the model: \( \text{DELETE}_N \) just means \( \text{WRITE}_N \) \( \downarrow \)

• Try adding this and run some tests…
Still not quite right...

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
<th>Client 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WRITE</strong> ‘a’ → ⊥</td>
<td><strong>READ</strong> → ‘a’</td>
<td><strong>WRITE</strong> ‘b’ → ‘a’</td>
</tr>
<tr>
<td><strong>WRITE</strong> ⊥ → ‘a’</td>
<td><strong>READ</strong> → ⊥</td>
<td></td>
</tr>
<tr>
<td><strong>READ</strong> → ‘b’</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write “a” on client 1
Delete the file
We now observe "b", so the stable value on the server must have been overwritten, despite the fact that ‘b’ was in conflict

Client 2 sees 1’s value
2 sees “missing” (so stable value at server is “missing”)

Now client 3 writes "b". Observes previous value ‘a’ (n.b.: not ⊥).
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
Precondition: $\text{Clean}_N = \text{DIRTY}$

Effect: $\text{Clean}_N \leftarrow \text{CLEAN}$

if $\text{Fresh}_N = \text{FRESH}$ then
  if $\text{LocalVal}_N \neq \text{ServerVal}$ then
    $\text{Fresh}_N' \leftarrow \text{STALE}$ for all $N' \neq N$
    $\text{ServerVal} \leftarrow \text{LocalVal}_N$
  else
    if $\text{LocalVal}_N \not\in \{\text{ServerVal}, \bot\}$ then
      $\text{Conflicts} \leftarrow \text{Conflicts} \cup \{\text{LocalVal}_N\}$
Surprises...
**Surprise:** Dropbox can (briefly) delete a newly created file…

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Create file</strong></td>
<td><strong>Create it again</strong></td>
</tr>
<tr>
<td><strong>Delete it</strong></td>
<td><strong>Observe creation</strong></td>
</tr>
<tr>
<td><strong>Create it again</strong></td>
<td><strong>File is gone!</strong></td>
</tr>
</tbody>
</table>

- **WRITE** ‘a’ → ⊥
- **WRITE** ⊥ → ‘a’
- **WRITE** ‘c’ → ⊥
- **READ** → ⊥
- **WRITE** ‘b’ → ‘a’

**Timing is critical!** Add **SLEEP** operations in tests
**Surprise:** Dropbox can (permanently) re-create a deleted file…

<table>
<thead>
<tr>
<th>Client 1</th>
<th>(other clients idle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create file</td>
<td></td>
</tr>
<tr>
<td>Delete it</td>
<td></td>
</tr>
<tr>
<td>File is back!</td>
<td></td>
</tr>
<tr>
<td><strong>WRITE</strong> ( 'b' ) ( \rightarrow \bot )</td>
<td></td>
</tr>
<tr>
<td><strong>WRITE</strong> ( \bot ) ( \rightarrow \ 'b' )</td>
<td></td>
</tr>
<tr>
<td><strong>READ</strong></td>
<td>( \rightarrow \ 'b' )</td>
</tr>
</tbody>
</table>

(Again, timing is critical)
**Surprise:** Dropbox can lose data

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WRITE</strong> ‘a’ → ‘b’</td>
<td><strong>WRITE</strong> ‘b’ → ⊥</td>
</tr>
<tr>
<td><strong>READ</strong></td>
<td>‘a’</td>
</tr>
<tr>
<td><strong>STABILIZE</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{ (‘a’, ø),</td>
</tr>
<tr>
<td></td>
<td>(‘b’, ø) }</td>
</tr>
</tbody>
</table>

**Create file**

**Old value persists on client 2!**

**New value persists on client 1**

**Overwrite it**

(Again, timing is important)

Client 1 believes it is still Fresh, so if we later write a new value on client 2, it will silently overwrite client 1’s value and no conflict file will be created.
Wrapping up...
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!
Thank you!

(Any questions?)