Lineage-driven Fault Injection

Peter Alvaro
The legend of partial failure
Back in the good old days
Back in the good old days

Transparency

Redundancy

Physical volumes

Logical volumes
The future is disorder

- Data-intensive systems are increasingly distributed
- *Partial failure* is widespread in distributed systems
- Fault-tolerant code is hard to get right
- Composing FT components is hard too!
Motivation: Kafka replication bug

Three correct components:
1. Primary/backup replication
2. Timeout-based failure detectors
3. Zookeeper

One nasty bug:
Acknowledged writes are lost
‘Molly' witnesses the bug

Brief network partition

- Replica b
- Replica c
- Zookeeper
- Replica a
- Client

1 -> 2
m

2 -> 3
m

3 -> 4
w

4 -> 5
a

5
CRASHED

1

a becomes primary and sole replica

a ACKs client write

Data loss
Fault-tolerance: the state of the art

1. Bottom-up approaches (e.g. verification)

2. Top-down approaches (e.g. fault injection)
Fault-tolerance: the state of the art

1. Bottom-up approaches (e.g. verification)

2. Top-down approaches (e.g. fault injection)
Fault-tolerance: the state of the art

1. Bottom-up approaches (e.g. verification)

2. Top-down approaches (e.g. fault injection)
Fault-tolerance: the state of the art

1. Bottom-up approaches (e.g. verification)

2. Top-down approaches (e.g. fault injection)
Lineage-driven fault injection

**Goal**: best of both worlds

**Main idea**: fault-tolerance is *redundancy*.

Reify *computation as data* using *lineage*
Broadcast protocols
Data lineage

\[ \log(B, \text{data})\]

\[ \log(B, \text{data}) \]

\[ \log(B, \text{data}) \]

\[ \log(B, \text{data}) \]

\[ \text{bcast}(A, \text{data}) \]
Executions provide “proofs” of outcomes

\[ AB_1 \xrightarrow{\text{node}(A, B)@1} \xrightarrow{\log(B, \text{data})@2} \xrightarrow{\log(B, \text{data})@3} \xrightarrow{\log(B, \text{data})@4} \xrightarrow{\log(B, \text{data})@5} \]

(this required a message from A to B at time 1)
Counterexample

Process b  Process a  Process c

1

log (LOST)

log

2
Sender retries

```
Sender retries

Process b  Process a  Process c

1

2

3

4

5

log  log  log

log  log  log

log  log  log

log  log  log
```
Sender retries

Retry provides redundancy in time

\[
\begin{align*}
\log(B, data)@5 \\
\log(B, data)@4 & \quad \log(A, data)@4 \\
\log(B, data)@3 & \quad \log(A, data)@3 \\
\log(B, data)@2 & \quad \log(A, data)@2 \\
\log(A, data)@1
\end{align*}
\]
Traces are forests of proof trees

\[ AB_1 \frac{\log(A, \text{data})@1}{\log(B, \text{data})@2} \frac{\text{node}(A, B)@1}{\log(B, \text{data})@3} \frac{\log(B, \text{data})@4}{\log(B, \text{data})@5} \frac{\log(A, \text{data})@2}{r_2} \frac{\log(B, \text{data})@3}{r_1} \frac{\log(B, \text{data})@4}{r_1} \frac{\log(B, \text{data})@5}{r_1} \]
Traces are forests of proof trees
Counterexample

1

log (LOST)

CRASHED

log

2
Symmetric retry

![Symmetric retry diagram]

- Process b
- Process a
- Process c

1. Process b
2. Process a
3. Process c

1. log
2. log
3. log
4. log
5. log
Symmetric retry

Redundancy in space and time
Let’s reflect

**Intuition:**
Fault-tolerance is redundancy in space and time.

**Strategy:**
Reason *backwards* from outcomes using *lineage*
Lineage exposes redundancy of outcome support
Solving for faults

1. Break a proof by dropping any contributing message.
2. Find a set of failures that breaks all proofs of a good outcome.

\[(AB_1 \lor BC_2) \land (AC_1) \land (AC_2)\]

- Disjunction
- Conjunction of disjunctions (AKA CNF)
Progress

**Lineage-driven fault injection** [SIGMOD’15]
Joint work: Joshua Rosen, Joseph M. Hellerstein (UC Berkeley)

**Large-scale integration** [ongoing]
Joint work: Kolton Andrus (Netflix)
Shortcomings and opportunities

1. Dependence on a custom language
2. Unavailability of fine-grained lineage
Proposal: lineage collection for service-oriented architectures
Shortcomings and opportunities

1. Dependence on a custom language
2. Unavailability of fine-grained lineage
3. No prioritization of fault space search
Proposal: fault selection as an optimization problem

“falsify all proofs” is a *minimal hitting set* problem.

\[(A \lor B \lor C) \land (C \lor D \lor E \lor F) \land (D \lor E \lor F \lor G) \land (H \lor I)\]

\[(A, B, C), (C, D, E, F), (D, E, F, G), (H, I)\]

E.g.: (C, E, H)

The size of this set is a measure of the implicit redundancy of the system.
Measuring fault-tolerance by counting alternatives
Measuring fault-tolerance by counting alternatives
Weights capture failure domains
Weights capture failure domains
Marching orders

1. Fine-grained lineage collection infrastructure for service-oriented architectures
   1. Leverage experience and contacts at Netflix
   2. Develop a shim layer at REST API boundaries

2. Fault selection as an optimization problem
   1. Leverage state-of-the-art ILP solvers
   2. Use probability distributions to provide weights for fault hierarchy
New project team

1. Peter Alvaro, Assistant Professor of CS
2. Wang-Chiew Tan, Professor of CS
3. Kamala Ramasubramanian, 1st year grad
4. [Student X]
5. [Student Y]
Lineage-driven fault injection

LDFI finds fault-tolerance violations quickly or guarantees that none exist.

LDFI finds bugs by *explaining* good outcomes.

Then it explains the bugs.
Queries?
By injecting only “interesting” faults...

<table>
<thead>
<tr>
<th>Program</th>
<th>Counterexample</th>
<th>LOC</th>
<th>EOT</th>
<th>EFF</th>
<th>Crashes</th>
<th>Combinations</th>
<th>Random</th>
<th>Molly</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>exe</td>
<td></td>
</tr>
<tr>
<td>simple-deliv</td>
<td>Figure 4a</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>$4.10 \times 10^3$</td>
<td>4.08</td>
<td>2</td>
</tr>
<tr>
<td>retry-deliv</td>
<td>Figure 4b</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>$4.07 \times 10^4$</td>
<td>75.24</td>
<td>3</td>
</tr>
<tr>
<td>classic-deliv</td>
<td>Figure 4c</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>$2.62 \times 10^5$</td>
<td>116.16</td>
<td>5</td>
</tr>
<tr>
<td>2pc</td>
<td>Figure 8</td>
<td>16</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>24</td>
<td>5.48</td>
<td>2</td>
</tr>
<tr>
<td>2pc-ctp</td>
<td></td>
<td>25</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>36</td>
<td>8.56</td>
<td>3</td>
</tr>
<tr>
<td>3pc</td>
<td>Figure 9</td>
<td>24</td>
<td>9</td>
<td>7</td>
<td>1</td>
<td>$2.43 \times 10^{26}$</td>
<td>40.60</td>
<td>55</td>
</tr>
<tr>
<td>Kafka</td>
<td>Figure 10</td>
<td>18</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>$1.85 \times 10^{25}$</td>
<td>1183.12</td>
<td>38</td>
</tr>
</tbody>
</table>

Molly finds bugs quickly
By injecting only “interesting” faults...

<table>
<thead>
<tr>
<th>Program</th>
<th>Bound</th>
<th>Combinations</th>
<th>Executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>redun-deliv</td>
<td>11</td>
<td>8.07 X 10^{18}</td>
<td>11</td>
</tr>
<tr>
<td>ack-deliv</td>
<td>8</td>
<td>3.08 X 10^{13}</td>
<td>673</td>
</tr>
<tr>
<td>paxos-synod</td>
<td>7</td>
<td>4.81 X 10^{11}</td>
<td>173</td>
</tr>
<tr>
<td>bully-leader</td>
<td>10</td>
<td>1.26 X 10^{17}</td>
<td>2</td>
</tr>
<tr>
<td>flux</td>
<td>22</td>
<td>6.20 X 10^{76}</td>
<td>187</td>
</tr>
</tbody>
</table>

Molly provides guarantees that outcomes are fault-tolerant.
Lineage-driven fault injection

**Approach**: think **backwards** from outcomes
Use **lineage** to find evidence of redundancy

**Original question**:  
• Could a bad thing ever happen?

**Reframed question**:  
• Why did a good thing happen?  
• What could have gone wrong?