FATE and DESTINI
A Framework for Cloud Recovery Testing

Haryadi S. Gunawi, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, and Koushik Sen

Thanh Do, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Dhruba Borthakur
Cloud and failure recovery

- Cloud
  - Thousands of commodity machines
  - “Rare (HW) failures become frequent” [Hamilton]

- Failure recovery
  - “… has to come from the software” [Dean]
  - “… must be a first-class op” [Ramakrishnan et al.]
  - But … hard to get right
Cloudy with a chance of failure

More in literature:
- Data loss, whole-system down in Google Chubby [Burrows06]
- 91 recovery issues found in HDFS over 4 years
- ...
Why?

- Testing is not advanced enough
  - Cloud systems face complex multiple, diverse failures

- Recovery is under-specified
  - Lots of custom recovery
  - Implementation is complex

- Need two advancements:
  - Exercise complex failure modes
  - Write recovery specifications and test the implementation
Cloud testing

FATE
Failure Testing Service

DESTINI
Declarative Testing Specifications
Contributions

- **FATE**
  - Exercise multiple, diverse failures
    - Over 40,000 unique combinations (80 hours)
    - Challenge: combinatorial explosion of multiple failures
  - Pruning strategies for failure exploration
    - An order of magnitude speedup
    - Found the same #bugs

- **DESTINI**
  - Facilitate recovery specifications
    - Reliability and availability related
  - Clear and concise (use Datalog, 5 lines/check)
  - Design patterns
Summary of results

- **Target 3 cloud systems**
  - HDFS (primary target), Cassandra, and ZooKeeper

- **HDFS recovery bugs**
  - Found 16 new bugs (+6 in newest)

- **Problems found**
  - Data loss
    - Buggy recovery wipes out all replicas
  - Unavailability
    - Broken rack-aware policy
    - Can’t restart after failures
Outline

- Introduction
- **FATE**
  - Failure IDs: abstraction for failure exploration
  - Pruning strategies
- **DESTINI**
- Evaluation
- Conclusion
HadoopFS (HDFS) Write Protocol

No failures

Setup Stage

Alloc Req

Data Transfer

Setup Recovery:
Recreate fresh pipeline \((1, 2, 4)\)

Data Transfer Recovery:
Continue on surviving nodes \((1, 2)\)
Failures and FATE

- **Failures**
  - **Anytime**: different stages $\rightarrow$ different recovery
  - **Anywhere**: N2 crash, and then N3
  - **Any type**: bad disks, partitioned nodes/racks

- **FATE**
  - Systematically exercise multiple, diverse failures
  - How? need to “remember” failures – via failure IDs
Abstraction of I/O failures

Building failure IDs
- Intercept every I/O
- Inject possible failures
  - Ex: crash, network partition, disk failure (LSE/corruption)

<table>
<thead>
<tr>
<th>I/O information:</th>
<th>OutputStream.read() in BlockReceiver.java</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;stack trace&gt;</td>
</tr>
<tr>
<td></td>
<td>Net I/O from N3 to N2</td>
</tr>
<tr>
<td></td>
<td>“Data Ack”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Injected failure:</th>
<th>Crash After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure ID: 2573</td>
<td></td>
</tr>
</tbody>
</table>

Note: FIDs A, B, C, ...
Brute-force exploration

Exp #1: A

Exp #2: B

Exp #3: C

1 failure / run

2 failures / run
Outline

- Introduction
- **FATE**
  - Failure IDs: abstraction of failures
  - Pruning strategies for failure exploration
- DESTINI
- Evaluation
- Conclusion
Combinatorial explosion

- Exercised over 40,000 unique combinations of 1, 2, and 3 failures per run
  - 80 hours of testing time!

**New challenge:**
Combinatorial explosion of multiple failures

```
A1  A2
A1  B2
B1  A2
B1  B2
...
```

Diagram showing connections between combinations A1, A2, A3, B1, B2, B3 with 2 failures per run.
Pruning multiple failures

- Properties of multiple failures
  - Pairwise dependent failure IDs
  - Pairwise independent failure IDs

- **Goal**: exercise distinct recovery behaviors
  - **Key**: some failures result in similar recovery
  - Result: > 10x faster, and found the same bugs
Dependent failures

- **Failure dependency graph**
  - Inject single failures first
  - Record subsequent dependent IDs
    - Ex: X depends on A
  - **Brute-force**: AX, BX, CX, DX, CY, DY

- **Recovery clustering**
  - Two clusters: \{X\} and \{X, Y\}

- **Only exercise distinct clusters**
  - Pick a failure ID that triggers a recovery cluster
  - Results: AX, CX, CY
Independent failure IDs

- Independent combinations
  - Ex: FP = 2, N = 3
  - \( FP^2 \times N (N - 1) \)

- Symmetric code
  - Just pick two nodes
  - \( N (N - 1) \rightarrow 2 \)
  - \( FP^2 \times 2 \)
Independent failure IDs

- **FP² bottleneck**
  - Ex: FP = 4
  - Real example: FP = 15

- **Recovery clustering**
  - Cluster A and B if: 
    fail(A) == fail(B)
  - Reduce **FP²** to **FP²_clumped**
  - E.g. 15 FPs to 8 FPs_clumped
FATE Summary

- Contributions
  - Exercise multiple, diverse failures (via failure IDs)
  - Pruning strategies (> 10x improvement)

- Limitations
  - I/O reordering
  - Inclusion of states to failure IDs
  - More failure modes
    - Transient, slow-down, and data-center partitioning
Outline

- Introduction
- FATE
- DESTINI: Declarative Testing Specifications
- Evaluation
- Conclusion
DESTINI: declarative specs

- Is the system correct under failures?
  - Need to write specifications
  - FATE needs DESTINI

[It is] great to document (in a spec) the HDFS write protocol ...

..., but we shouldn't spend too much time on it, ... a formal spec may be overkill for a protocol we plan to deprecate imminently.
Declarative specs

- How to write specifications?
  - Developer friendly (clear, concise, easy)

- Datalog: a declarative relational logic language
  - Easy to express logical relations
  - (just for writing specifications)
How to write specs?
- Violations
- Expectations
- Facts

How to write recovery specs?
- “... recovery is under specified” [Hamilton]
- Precise failure events
- Precise check timings

How to test implementation?
- Interpose I/O calls (lightweight)
- Deduce expectations and facts from I/O events
Specification template

“Throw a violation if an expectation is different from the actual behavior”

violationTable(…) :-
expectationTable(…),
NOT-IN actualTable(…)

Datalog syntax:

head() :- predicates(), …
:- derivation
, AND
incorrectNodes (Block, Node)

expectedNodes (Block, Node)

actualNodes (Block, Node)

incorrectNodes(B, N) :- expectedNodes(B, N), NOT-IN actualNodes(B, N);

“Block replicas should exist in surviving nodes”
incorrectNodes(B, N) :- expectedNodes(B, N), NOT-IN actualNodes(B, N);
Building expectations

- Ex: which nodes **should** have the blocks?
  - Deduce expectations from *I/O events* (italic)

<table>
<thead>
<tr>
<th>expectedNodes</th>
<th>(Block, Node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Node 1</td>
</tr>
<tr>
<td>B</td>
<td>Node 2</td>
</tr>
<tr>
<td>B</td>
<td>Node 3</td>
</tr>
</tbody>
</table>

expectedNodes (B, N) :-
getBlockPipe (B, N);

#1: incorrectNodes(B, N) :- expectedNodes(B, N), NOT-IN actualNodes(B, N);
#1: incorrectNodes(B, N) :- expectedNodes(B, N), NOT-IN actualNodes(B, N);
#2: expectedNodes(B, N) :- getBlockPipe(B,N);

DEL expectedNodes (B, N) :-
expectedNodes (B, N),
fateCrashNode (N)

DESTINI needs FATE
Precise failure events

<table>
<thead>
<tr>
<th>Event</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td><code>incorrectNodes(B,N) :- expectedNodes(B,N), NOT-IN actualNodes(B,N)</code></td>
</tr>
<tr>
<td>#2</td>
<td><code>expectedNodes(B,N) :- getBlockPipe(B,N);</code></td>
</tr>
<tr>
<td>#3</td>
<td><code>expectedNodes(B,N) :- expectedNodes(B,N), fateCrashNode(N), writeStg(B,Stage), Stage == “DataTr”</code></td>
</tr>
<tr>
<td>#4</td>
<td><code>writeStg(B,“DataTr”) :- writeStg(B,“Setup”), nodesCnt(Nc), acksCnt (Ac), Nc==Ac</code></td>
</tr>
<tr>
<td>#5</td>
<td><code>nodesCnt (B, CNT&lt;N&gt;) :- pipeNodes (B, N);</code></td>
</tr>
<tr>
<td>#6</td>
<td><code>pipeNodes (B, N) :- getBlockPipe (B, N);</code></td>
</tr>
<tr>
<td>#7</td>
<td><code>acksCnt (B, CNT&lt;A&gt;) :- setupAcks (B, P,“OK”);</code></td>
</tr>
<tr>
<td>#8</td>
<td><code>setupAcks (B, P,A) :- setupAck (B, P,A);</code></td>
</tr>
</tbody>
</table>

DEL `expectedNodes (B, N), fateCrashNode (N), writeStg (B, Stage), Stage == “Data Transfer”;`
Violation and check-timing

#1:
incorrectNodes(B, N) :-
  expectedNodes(B, N),
  NOT-IN actualNodes(B, N),
  completeBlock (B);

- **Recovery ≠ invariant**
  - If recovery is ongoing, invariants are violated
  - Don’t want false alarms

- **Need precise check timings**
  - Ex: *upon block completion*
DESTINI Summary

- **Support recovery specs**
  - Reliability and availability related
  - Clear and concise (use Datalog)

- **Design patterns**
  - Add detailed specs
  - Write specs from different views (global, client, ...)
  - Incorporate diverse failures (crashes, rack partitions)
  - ... more in the paper
Outline

- Introduction
- FATE
- DESTINI
- Evaluation and conclusion
Evaluation

- Implementation complexity
  - ~6000 LOC in Java

- Target 3 popular cloud systems
  - HDFS (primary), ZooKeeper, Cassandra

- HDFS recovery bugs
  - Found 22 new bugs
    - 8 bugs due to multiple failures
    - Data loss, unavailability bugs
  - Reproduced 51 old bugs
If multiple racks are available (reachable), a block should be stored in a minimum of two racks.

Availability bug!

#replicas = 3, locations are not checked, B is not migrated to R2

FATE injects rack partitioning
### Availability bug

“If multiple racks are available (reachable), a block should be stored in a minimum of two racks”

```prolog
errorSingleRack(B) :- rackCnt(B,Cnt), Cnt==1, blkRacks(B,R), connected(R,Rb), endOfReplicationMonitor (\_);
```

<table>
<thead>
<tr>
<th>errorSingleRack</th>
<th>rackCnt</th>
<th>blkRacks</th>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B, 1</td>
<td>B, R1</td>
<td>R1, R2</td>
</tr>
</tbody>
</table>
Pruning Efficiency

- Reduce #experiments by an order of magnitude
  - Each experiment = 4-9 seconds

- Found the same number of bugs
  - (by experience)

Bar chart:
- Brute Force: 7720
- Pruned: 618
- Write + 2 crashes: 5000
- Append + 2 crashes: 5000
- Write + 3 crashes: 7720
- Append + 3 crashes: 618
### Specification simplicity

<table>
<thead>
<tr>
<th>Framework</th>
<th>#Chks</th>
<th>Lines/Chk</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3S [NSDI ’08]</td>
<td>10</td>
<td>53</td>
</tr>
<tr>
<td>Pip [NSDI ’06]</td>
<td>44</td>
<td>43</td>
</tr>
<tr>
<td>WiDS [NSDI ’07]</td>
<td>15</td>
<td>22</td>
</tr>
<tr>
<td>P2 Monitor</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>DESTINI</td>
<td>74</td>
<td>5</td>
</tr>
</tbody>
</table>

- Compared to other related work
Conclusion

- Cloud software systems
  - Must deal with HW failures

- FATE and DESTINI
  - Explore multiple, diverse failures systematically
  - Facilitate concise recovery specifications
  - A unified framework
    - FATE needs DESTINI
    - DESTINI needs FATE

- Real-world adoption in progress
Thank you! Questions?

http://boom.cs.berkeley.edu  http://cs.wisc.edu/adsl