HARDFS:  
Hardening HDFS with  
Selective and Lightweight Versioning

Thanh Do, Tyler Harter, Yingchao Liu,  
Andrea C. Arpaci-Dusseau,  
and Remzi H. Arpaci-Dusseau

Haryadi S. Gunawi
Cloud Reliability

- Cloud systems
  - Complex software
  - Thousands of commodity machines
  - “Rare failures become frequent” [Hamilton]

- Failure detection and recovery
  - “… has to come from the software” [Dean]
  - “… must be a first-class operation” [Ramakrishnan et al.]
Fail-stop failures

- Machine crashes, disk failures
- Pretty much handled
- Current systems have sophisticated crash-recovery machineries
  - Data replication
  - Logging
  - Fail-over
Fail-silent failures

- Exhibits incorrect behaviors instead of crashing
- Caused by memory corruption or software bugs
- Crash recovery is useless if fault can spread
Fail-silent failure headlines

S3 Outage Highlights Fragility of Web Services

Gmail data loss bug causes complete data loss, calls for tape backups

By Chris Alexander, published on 02 Mar 2011 | Filed in Security, Azure
Current approaches

Replicated state machine using BFT library

• High resource consumption
• High engineering effort
• Rare deployment

N-Version programming

Ver. 1

Agree?

Ver. 2

Ver. 3
Selective and Lightweight Versioning (SLEEVE)

- 2nd version models basic protocols of the system
- Detects and isolates fail-silent behaviors
- Exploits crash recovery machinery for recovery
Selective and lightweight versioning (SLEEVE)

- **Selective**
  - Goal: small engineering effort
  - Protects important parts
    - Bug sensitive
    - Frequently changed
    - Currently unprotected

- **Lightweight**
  - Avoids replicating full state
  - Encodes states to reduce space
HARDFS

- HARDFS - hardened version HDFS:
  - Namespace management
  - Replica management
  - Read/write protocol

- HARDFS detects and recovers from:
  - 90% of the faults caused by random memory corruption
  - 100% of the faults caused by targeted memory corruption
  - 5 injected software bugs

- Fast recovery using micro-recovery
  - 3 orders of magnitude faster than full reboot

- Little space and performance overhead
Outline

✓ Introduction

☐ HARDFS Design

☐ HARDFS Implementation

☐ Evaluation

☐ Conclusion
Case study: namespace integrity

Normal Operation

Client → NameNode: Create(F) → txCreat(F) → F

Corrupted HDFS

Client → NameNode: exists(F) = No

HARDFS

Client → NameNode: exists(F) = Yes

Incorrect behavior

Trusted source

F

F

F
SLEEVE layer components

- Interposition module
- State manager
- Action verifier
- Recovery module
State manager

- Replicates *subset* of state of the main version
  - Directory entries without modification time

- Adds new state *incrementally*
  - Adds permissions for security checks

- Understands *semantics* of various protocol messages and thread events to update state correctly

- **Compresses** state using compact encoding
Naïve: Full replication

- HDFS master manages millions of files
- 100% memory overhead reduces HDFS master scalability
Lightweight: Counting Bloom Filters

- Space-efficient data structure

- Supports 3 APIs
  - `insert("A fact")`
  - `delete("A fact")`
  - `exists("A fact")`
Lightweight: Counting Bloom Filters

- Suitable for boolean checking
  - Does F exist?
  - Does F has length X?
  - Has block B been allocated?

“F is 10 bytes”

insert(“F is 10 bytes”)

“Give me length of F”

exists(“F is 5 bytes”) → NO

Disagreement detected!
Challenges of using Counting Bloom Filters

- Hard to check stateful system
- False positives
Non-boolean verification

“F is 20 bytes”

Bloom filter does not support this API

\[ X = \text{returnSize}(F) \]
\[ \text{delete}(F:X) \]
\[ \text{insert}(F:20) \]
Non-boolean verification

“F is 20 bytes”

Ask-Then-Check

X \leftarrow \text{MainVersion.returnSize}(F);
\text{IF exists}(F:X)
  \text{delete}(F:X);
  \text{insert}(F:20);
\text{ELSE}
  \text{initiate recovery};
Stateful verification

- Ask Then Check
- Checking stateful systems
- Bloom Filter (boolean verification)
Dealing with False positive

- Bloom filters can give false positive
  - 4 per billion
  - 1 false positive per month (given 100 op/s)
- Only leads to unnecessary recovery
Outline

✓ Introduction

☐ HARDFS Design
  ✓ Lightweight
    ▪ Selective
    ▪ Recovery

☐ HARDFS Implementation

☐ Evaluation

☐ Conclusion
Selective Checks

- **Goals:** small engineering effort
- **Selectively chooses namespace protection**
- **Excludes security checks**

The diagram illustrates the process of handling disagreements in a file system. A client attempts to create a file, and the HDFS master checks if the file already exists. If the client operation log indicates a disagreement, it triggers a function to handle the disagreement:

```
X ← mainVersion.exists(F);
Y ← bloomFilter.exists(F);
If X != Y then
    handleDisagreement();
```

- **create(F)** from Client
- **exists(F)** from HDFS Master
- **txCreate** from HDFS Master to Client
- **Operation log**
- **Disagreement detected!**
Incorrect action examples

Normal correct action
Create(F) → txCreate(F)

Corrupt action
Create(F) → reject

Missing action
Create(F) → Mkdir(D)

Orphan action
Create(F) → txCreate(F)

Out-of-order action
Create(D/F) → Mkdir(D)

txCreate(D/F)

txMkdir(D)

All of these happen in practice
**Action verifier**

- Set of *micro-checks* to detect incorrect actions of the main version

- **Mechanisms:**
  - Expected-action list
  - Actions dependency checking
  - Timeout
  - Domain knowledge to handle disagreement
Outline

✓ Introduction

☐ HARDFS Design
  ✓ Lightweight
  ✓ Selective
  ☐ Recovery

☐ HARDFS Implementation

☐ Evaluation

☐ Conclusion
Recovery

- Crash is good provided no fault propagation
- Detects and turns bad behaviors into crashes
- Exploits HDFS crash recovery machineries
HARDFS Recovery

- Full recovery (crash and reboot)
- Micro-recovery
  - Repairing the main version
  - Repairing the 2nd version
Crash and Reboot

- Full state is reconstructed from trusted sources
- Full recovery may be **expensive**
  - Restarting an HDFS master could take **hours**
Micro-recovery

- Repairs only corrupted state from trusted sources
- Falls back to full reboot when micro-recovery fails
Repairing main version

Main Version

F:200

2nd Version

F:100

Trusted source: checkpoint file

Direct update

F:200 ← F:100
Repairing 2\textsuperscript{nd} version

Main Version
F: 100

2\textsuperscript{nd} Version
F: F000

Must:
1. Delete(“F is 200 bytes”)
2. Insert(“F is 100 bytes”)

Solution:
1. Start with an empty BF
2. Add facts as they are verified

Trusted source: checkpoint file
Outline

✓ Introduction
✓ HARDFS Design

☐ HARDFS Implementation
☐ Evaluation
☐ Conclusion
Implementation

- Hardens three functionalities of HDFS
  - Namespace management (HARDFS-N)
  - Replica management (HARDFS-R)
  - Read/write protocol of datanodes (HARDFS-D)

- Uses 3 Bloom filters API
  - `insert(“a fact”)`, `delete(“a fact”)`, `exists(“a fact”)`

- Uses `ask-then-check` for non-boolean verification
Protecting namespace integrity

- Guards namespace structures necessary for reaching data:
  - File hierarchy
  - File-to-block mapping
  - File length information

- Detects and recovers from namespace-related problems:
  - Corrupt file-to-block mapping
  - Unreachable files
Namespace management

<table>
<thead>
<tr>
<th>Message</th>
<th>Logic of the secondary version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create(F): Client request NN to create F</td>
<td><strong>Entry:</strong> If <em>exists</em>(F) Then reject; Else <em>insert</em>(F); generateAction(txCreate[F]); <strong>Return</strong>: check response;</td>
</tr>
<tr>
<td>AddBlock(F): client requests NN to allocate a block to file F</td>
<td><strong>Entry:</strong> F:X = <em>ask-then-check</em>(F); <strong>Return:</strong> B = addBlk(F); If <em>exists</em>(F) &amp; !<em>exists</em>(B) Then X' = X ∪ {B}; <em>delete</em>(F:X); <em>insert</em>(F:X'); <em>insert</em>(B@0); Else declare error;</td>
</tr>
</tbody>
</table>
Outline

✓ Introduction
✓ HARDFS Design
✓ HARDFS Implementation
✓ Evaluation and Conclusion
Is HARDFS robust against fail-silent faults?

How much time and space overhead incurred?

Is micro-recovery efficient?

How much engineering effort required?
# Random memory corruption results

<table>
<thead>
<tr>
<th>Outcome</th>
<th>HDFS</th>
<th>HARDFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Silent failure</td>
<td>117</td>
<td>9</td>
</tr>
<tr>
<td>Detect and reboot</td>
<td>-</td>
<td>140</td>
</tr>
<tr>
<td>Detect and micro-recover</td>
<td>-</td>
<td>107</td>
</tr>
<tr>
<td>Crash</td>
<td>133</td>
<td>268</td>
</tr>
<tr>
<td>Hang</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>No problem observed</td>
<td>728</td>
<td>460</td>
</tr>
</tbody>
</table>

- # fail-silent failures reduced by factor of 10
- Crash happens twice as often
# Silent failures

<table>
<thead>
<tr>
<th>FIELD</th>
<th>HDFS</th>
<th>HARDFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathname</td>
<td>95</td>
<td>0</td>
</tr>
<tr>
<td>replication</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>modification time</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>permission</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>block size</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>
HARDFS with Bloom filter incurs little space overhead (2.6%)
Recovery Time

<table>
<thead>
<tr>
<th>Recovery Time (seconds)</th>
<th>File system size (number of files)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200K</td>
</tr>
<tr>
<td></td>
<td>400K</td>
</tr>
<tr>
<td></td>
<td>600K</td>
</tr>
<tr>
<td></td>
<td>800K</td>
</tr>
<tr>
<td></td>
<td>1000K</td>
</tr>
</tbody>
</table>

- Rebooting NameNode is expensive
- Micro-recovery is 3 order of magnitude faster
## Complexity (LOC)

<table>
<thead>
<tr>
<th>Functionality</th>
<th>HDFS</th>
<th>HARDFS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Namespace management</td>
<td>10114</td>
<td>1751</td>
<td>17%</td>
</tr>
<tr>
<td>Replica management</td>
<td>2342</td>
<td>934</td>
<td>40%</td>
</tr>
<tr>
<td>Read/write protocol</td>
<td>5050</td>
<td>944</td>
<td>19%</td>
</tr>
<tr>
<td>Others</td>
<td>13339</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- Lightweight versions are smaller
# Injecting software bugs

<table>
<thead>
<tr>
<th>Bug</th>
<th>Year</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HADOOP-1135</td>
<td>2007</td>
<td>Major</td>
<td>Blocks in block report wrongly marked for deletion</td>
</tr>
<tr>
<td>HADOOP-3002</td>
<td>2008</td>
<td>Blocker</td>
<td>Blocks removed during safemode</td>
</tr>
<tr>
<td>HDFS-900</td>
<td>2010</td>
<td>Blocker</td>
<td>Valid replica deleted rather than corrupt replica</td>
</tr>
<tr>
<td>HDFS-1250</td>
<td>2010</td>
<td>Major</td>
<td>Namenode processes block report from dead datanode</td>
</tr>
<tr>
<td>HDFS-3087</td>
<td>2012</td>
<td>Critical</td>
<td>Decommission before replication during namenode restart</td>
</tr>
</tbody>
</table>
Conclusion

- Crashing is good
- To die (and be reborn) is better than to lie
- But lies do happen in reality
- HARDFS turns lies into crashes
- Leverages existing crash recovery techniques to resurrect
Thank you!
Questions?

http://research.cs.wisc.edu/adsl/
http://wisdom.cs.wisc.edu/
http://ucare.cs.uchicago.edu/