Understanding and Detecting Software Upgrade Failures in Distributed Systems

Huangxing Chen

Background Introduction

Intro

When we build our services on distributed system, we will want upgrading for:

- Adding new features
- Improve performance
- ...

As for the upgrading:

- Upgrading is unavoidable and will is done frequently.
- Most of the severe outages and failures are from system upgrading(e.g. Azure and Dropbox outage ...)
- There are some special **failures** that will **only show up when upgrading**, specifically

Motivation

Upgrade failures are problematic because:

- They are not caused simply by code bugs or misconfiguration
- They may have large scale influence that can paralyze the whole system
- System is vulnerable when updating and failures can greatly affect the service quality
- They can lead to data and system state corruption
- Difficult to run test case for the upgrade process
- Failure data is hard to collect

Goals

- Prove that most of the upgrade failures are with high severity and hard to be caught before being released to the public
- Find out the root causes and methods to avoid them
- Define the trigger conditions of failures as a guidance to develop testing framework and cases
- Build tools to tackle these failures

Methodology

Studied 123 kinds of reported and resolved failures from 8 data-intensive systems(e.g HDFS, Hadoop MapReduce framework...)

There are limitations:

- Representativeness of Reports and Distributed System
- The filtering criteria of the failure reports
- Observer errors(minimized by cross-inspection)

In-depth Analysis

Severity & Root cause & Trigger condition

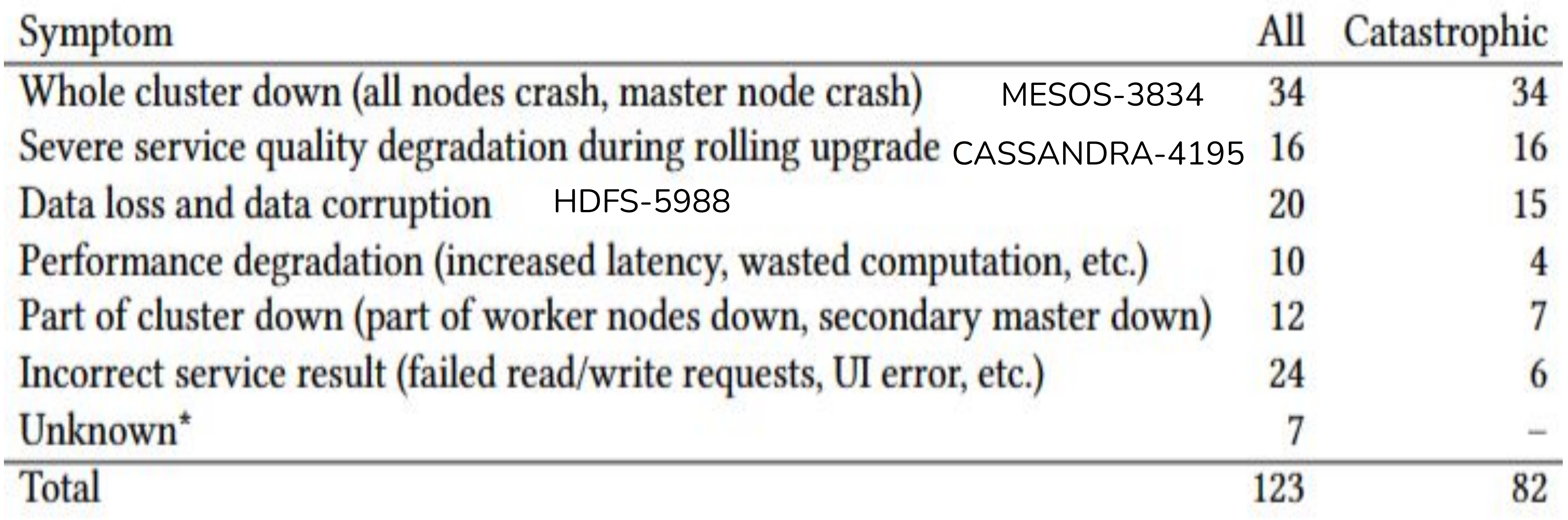
Severity Study

Severity of upgrade failures vs non-upgrade failures:

	Cassandra		Other Systems	
	Upgrade failures	Non-upgrade failures	Upgrade failures	Non-upgrade failures
Portion of high priority bugs	53%	20%	93%	59%

Severity Study

The high severity comes from the symptoms:



Symptom		All	Catastrophic
Whole cluster down (all nodes crash, master node crash)	MESOS-3834	34	34
Severe service quality degradation during rolling upgrade	CASSANDRA-4195	16	16
Data loss and data corruption	HDFS-5988	20	15
Performance degradation (increased latency, wasted computation, etc.)		10	4
Part of cluster down (part of worker nodes down, secondary master down)		12	7
Incorrect service result (failed read/write requests, UI error, etc.)		24	6
Unknown*		7	–
Total		123	82

Severity Findings

- Upgrade failures have significantly higher priority than regular failures.
- The majority (67%) of upgrade failures are catastrophic
- Most (70%) upgrade failures have easy-to-observe symptoms like node crashes or fatal exceptions
- The majority (63%) of upgrade bugs were not caught before code release

Root Cause Study

Four types:

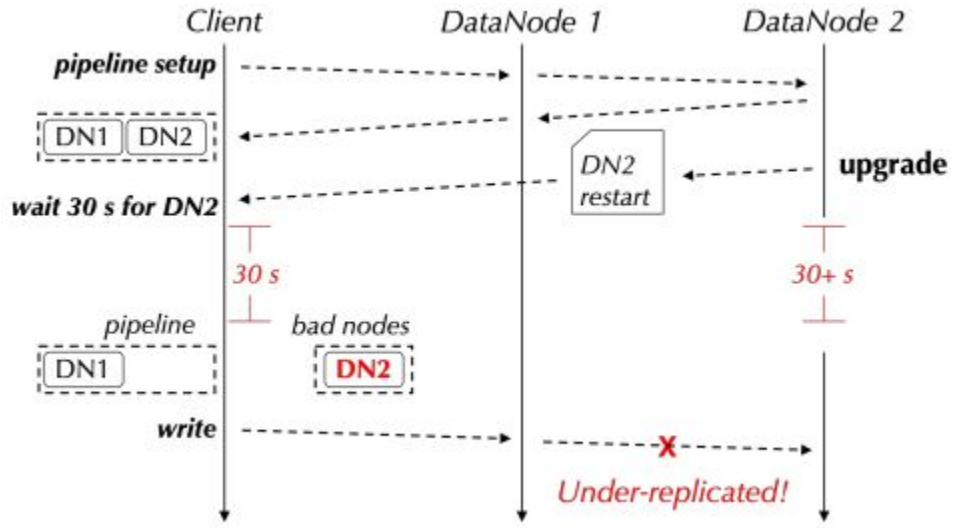
- Incompatible cross-version interaction(63%)
- Broken upgrade operation(33%)
- Misconfiguration(3%)
- Broken library dependency(2%)

Incompatible cross-version interaction

- Data source:
  - Persistent storage data
  - Network messages
- Incompatibility type:
  - Syntax
    - Serialization library data(Class-like)
    - Enum(Array-like)
    - System-specific data
  - Sematic(e.g. Different meaning of a “Default” setting)
    - The key part is version handling and checking
      - Authors provided suggestions and examples

Broken upgrade operation

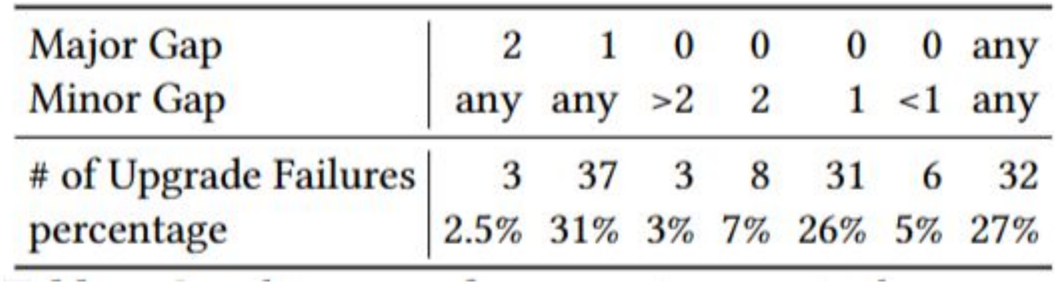
Definition: unexpected interaction between the upgrade operation and specific regular operations of the system



Triggers study

This study is to find out in which circumstances an upgrade failure are likely to happen, which helps providing opportunities for automated testing.

Problem: It’s too much to compare all N^2 combinations



Major Gap	2	1	0	0	0	0	any
Minor Gap	any	any	>2	2	1	<1	any
# of Upgrade Failures	3	37	3	8	31	6	32
percentage	2.5%	31%	3%	7%	26%	5%	27%

Note: The version number is in form of: <Major>.<Minor>.<bug-fix>

How to Trigger Failures

- All of the upgrade failures require no more than 3 nodes to trigger (Caused by Persistent data or communication)

- Close to 90% of the upgrade failures are deterministic, not requiring any special timing to trigger (An exception was mentioned before)

- Half of upgrade bugs can be triggered by stress testing operations with default configurations.
    - The others need special configurations and operations but most can be covered by using existing unit tests.

State of Art Testing

The 8 systems(i.e. Cassandra, Mesos,...) studied by the authors has testing scripts for upgrade operations but still had a majority of bugs caught after release.

Two Key Limitations:

- They used testing workload designed from scratch instead of the mature and much larger amount one for stress tests.
- The tests didn’t consider different situations about versions, configurations and upgrade scenarios.

New Testing And Detecting Tools

DUPTester & DUPChecker

Distributed system UPgrade Tester(DUPTester)

-to expose upgrade failures through in-house testing

Architecture:

- Pre-loaded system containers with different versions
- A 3-node cluster
- A shared directory to store persistent data for other containers’ accessing.

DUPTester

Upgrade Scenarios: Full-stop, Rolling, New node joining

Testing workload:

- Leveraging existing stress testing workload.
- For unit testing:
  - Translate them to client side command scripts(python programs, according to the authors)
  - Test their influence on system states(only for full-stop upgrade)

DUPTester

Evaluation:

- Tested 3 studied systems(Cassandra, HBase and Kafka) and 1 unstudied system(Hive)
- The version gap of upgrading are either 1-2 minor versions or 1 major version.

Result:

- Found 20 previously unknown failures(7 of them are confirmed by the developers)
- The triggering workloads and configurations are not covered by existing testing scripts of Cassandra

Distributed system UPgrade Checker(DUPChecker)

-to detect upgrade failures caused by data-syntax incompatibility through static program analysis

For two types of data:

- Serialization Libraries Data
  - Checker already exists
  - Creates a parser for protocol files to compare the data format of the same data member from different versions
- Enum Data
  - Checks whether the enum class has member addition or deletion across two versions.
  - If so, it considers it as a bug, otherwise a vulnerability to future changes

DUPChecker Results

It found 878 unknown incompatibilities of the first data type. According to the authors, there is no false positive.

And for the second type, 2 newly found bugs were confirmed and fixed by the developers and 3 of the 6 new vulnerabilities are confirmed and fixed.

Future Works

Suggested Research Direction

- Apply new techniques to explore the test space and trigger more upgrade failures.
- Developed more static analysis techniques to detect incompatibilities caused by changing file names, changing configurations….
- According to the analysis, applying flexible and efficient serialization libraries to more data will help eliminate upgrade failures.

Related Works

Studies on Upgrading Failures

- Liu et al.[1] pointed out that software upgrade is one of the reasons for incompatible data-formats but didn’t offer details and corresponding solutions.

- Gunawi et al.[2] found that 16% of cloud service outages involve hardware or software upgrade without in-depth analysis.

Studies on Upgrading Failures

- Tudor et al.[3] analyzed 55 upgrade failures from a e-commerce system, a database system, and Apache web server focusing on causes like misconfiguration, broken dependency, and operator error.

- Some studies[4-6] focus on the root cause of distributed system failures without discussion about upgrade failures.

- The authors are the first to focus on upgrade failures caused by software defects in distributed systems.

Review

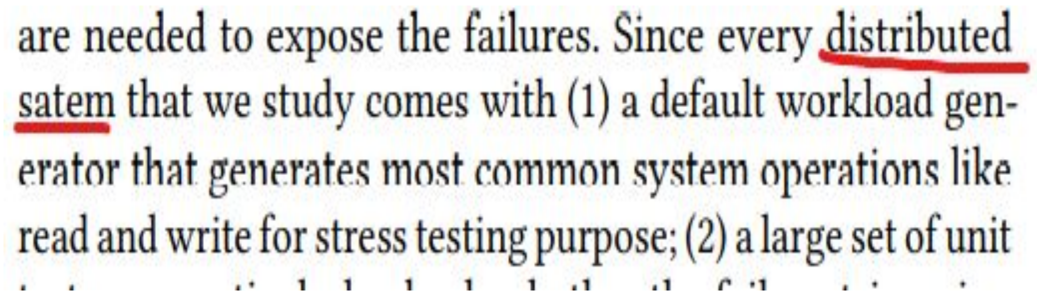
Strong Points:

- Provided in-depth analysis on the upgrading failures including severity, trigger conditions and root cause which is also inspiring for future studying.

- Developed powerful tools to test and detect upgrading failures which are experimentally proven to be more powerful than existing test scripts.

Weak Points

- Only one root cause are covered by the DUPChecker
- I will be hard for the DUPTester to cover most of the special circumstances if there are not enough existing test cases.
- And a typo in the article:


are needed to expose the failures. Since every distributed satem that we study comes with (1) a default workload generator that generates most common system operations like read and write for stress testing purpose; (2) a large set of unit

References

[1]Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS ’19, pages 155–162, 2019.

[2]Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC ’16, pages 1–16, 2016.

[3]Tudor Dumitraş and Priya Narasimhan. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, Middleware ’09, pages 18:1–18:20, 2009.

[4]Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI’14), pages 249–265, 2014.

[5]Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In Proceedings of the ACM Symposium on Cloud Computing, SOCC '14, pages 7:1–7:14, 2014

[6]A. Rabkin and R.H. Katz. How Hadoop clusters break. Software, IEEE, 30(4):88–94, 2013