An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems

Yu Gao, Wensheng Dou†
State Key Lab of Computer Science, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences, China
{gaoyu15, wsdou}@otcaix.iscas.ac.cn

Feng Qin
Dept. of Computer Science and Engineering
The Ohio State University, United States
qin.34@osu.edu

Chushu Gao, Dong Wang, Jun Wei†
State Key Lab of Computer Science, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences, China
{gaochushu, wangdong18, wj}@otcaix.iscas.ac.cn

Ruirui Huang, Li Zhou, Yongming Wu
Alibaba Group, China
{ruirui.huang, celly.zl, yongming.wym}@alibaba-inc.com

ABSTRACT

In large-scale distributed systems, node crashes are inevitable, and can happen at any time. As such, distributed systems are usually designed to be resilient to these node crashes via various crash recovery mechanisms, such as write-ahead logging in HBase and hinted handoffs in Cassandra. However, faults in crash recovery mechanisms and their implementations can introduce intricate crash recovery bugs, and lead to severe consequences.

In this paper, we present *CREB*, the most comprehensive study on 103 *C*rash *RE*covery *B*ugs from four popular open-source distributed systems, including ZooKeeper, Hadoop MapReduce, Cassandra and HBase. For all the studied bugs, we analyze their root causes, triggering conditions, bug impacts and fixing. Through this study, we obtain many interesting findings that can open up new research directions for combating crash recovery bugs.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; **Software reliability**; Software testing and debugging

KEYWORDS

Distributed systems, crash recovery bugs, empirical study

ACM Reference format:

Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236030

† Wensheng Dou and Jun Wei are the corresponding authors.

ESEC/FSE '18, November 4 – 9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11...$15.00
https://doi.org/10.1145/3236024.3236030

1 INTRODUCTION

Large-scale distributed systems have become pervasive and indispensable to our daily lives. While enterprises leverage distributed systems such as scalable computing frameworks [8], storage systems [4][9][13], cluster management services [21][26] and synchronization services [3], for cloud computing and big data analytics, consumers rely on distributed systems employed by large Internet companies such as Google and Alibaba to conveniently access popular services, e.g., online searching and shopping. Failures of such large-scale distributed systems can adversely impact billions of users and lead to huge financial losses [56][57]. Therefore, the reliability of distributed systems is of critical importance.

However, the reliability of large-scale distributed systems is threatened by node crashes -- a norm in these systems. Large-scale distributed systems are commonly built on farms of machines (nodes), comprising thousands of commodity machines. A machine (node) may suffer from power failures, hardware faults and software faults, and thus the node may become unavailable. As the number of nodes in a distributed system increases, node failures become normal. When a node crashes, the on-going tasks and in-memory data in the node are lost. Other live nodes in the cluster may not work properly especially when the crash node is the dominant node, e.g., the master node. To provide high reliability, the live nodes in the cluster should detect and handle node crashes in time. The crash node should correctly rejoin the cluster when it is rebooted. Therefore, automated recovery must be a first-class operation of distributed systems [6].

To combat node crashes, developers have introduced sophisticated crash recovery mechanisms into large-scale distributed systems, e.g., write-ahead logging in HBase [59] and hinted handoffs in Cassandra [55]. However, it is still challenging to handle node crashes in these distributed systems. Node crashes can happen unpredictably on any node at any time, and cause many kinds of crash scenarios, such as partially-updated persistent states and in-memory data loss. It is difficult for developers to contemplate all possible crash scenarios, and correctly implement corresponding crash recovery mechanisms. It is also impossible to inject crashes in all possible scenarios, and test a distributed system thoroughly [14]. As a result, inadequate crash recovery mechanisms and their

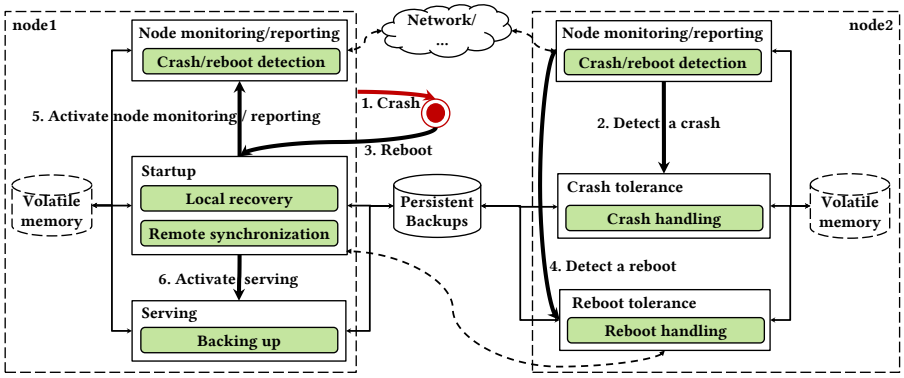

Figure 1: The general crash recovery model for distributed systems. *Steps 1-6 show a general crash recovery process. Green rounded rectangles denote seven crash recovery components. We use solid and thick arrows to denote that a node goes into a terminate state or starts up another execution process when encountering certain events. For example, when node1 encounters a crash event (step 1), node1 terminates immediately. When node2's crash detection component detects a crash (step 2), the crash recovery process is started. We use solid and thin arrows to denote data reading and writing, and dotted arrows to denote messages among nodes.*

incorrect implementations can introduce intricate crash recovery bugs, and lead to severe consequences. For example, in our study, 38% of our studied crash recovery bugs cause node downtimes (e.g., node hangs), and 21% of the bugs cause data-related failures. In this paper, crash recovery bugs refer to the bugs that exist in the crash recovery related mechanisms and implementations.

To better combat crash recovery bugs in distributed systems, our community urgently needs a thorough understanding of them. Crash recovery bugs have been widely studied on single-machine systems [5][16][34][41][44]. In distributed systems, existing techniques mainly focus on crash recovery bug detection through model checking and crash injection, e.g., MODIST [40], DEMETER [17], SAMC [27], PreFail [24], MOLLY [2], CORDS [12], PACE [1], FATE and DESTINI [14]. However, it is unknown whether there exist some patterns among crash recovery bugs. Although some empirical studies have been conducted for different kinds of bugs in distributed systems, e.g., CBS [15], TaxDC [28], and timeout issues [7]. But none of them specially dissects crash recovery bugs. We believe that an in-depth study of crash recovery bugs can further promote the reliability research in distributed systems.

In this paper, we present *CREB*, the first (to the best of our knowledge) comprehensive analysis of 103 *C*rash *RE*covery *B*ugs from four popular large-scale distributed systems, including ZooKeeper [49], Hadoop MapReduce [47], Cassandra [45] and HBase [48]. We thoroughly study these bugs and try to answer the following research questions.

- RQ1 (Root causes): What are the root causes for crash recovery bugs?
- RQ2 (Triggering conditions): How is a crash recovery bug triggered? How complicated is it to trigger a bug?
- RQ3 (Bug impacts): What impacts do crash recovery bugs have?
- RQ4 (Fixing): How do developers fix crash recovery bugs?

Through our in-depth analysis against the above four research questions, we obtain many interesting findings. We summarize our main findings as follows:

- Crash recovery bugs are caused by five types of bug patterns, i.e., incorrect backup (17%), incorrect crash / reboot detection (18%), incorrect state identification (16%), incorrect state recovery (28%) and concurrency (21%). These bug patterns motivate new approaches to detect crash recovery bugs. For example, we can detect incorrect backup bugs by analyzing whether in-memory data are backed up in all program paths.
- Almost all (97%) of crash recovery bugs involve no more than four nodes. This finding indicates that we can detect crash recovery bugs in a small set of nodes, rather than thousands.
- A majority (87%) of crash recovery bugs require a combination of no more than three crashes and no more than one reboot. It suggests that we can systematically test almost all node crash scenarios with very limited crashes and reboots.
- Crash recovery bugs are difficult to fix. 12% of the fixes are incomplete, and 6% of the fixes only reduce the possibility of bug occurrence. This indicates that new approaches to validate crash recovery bug fixes are necessary.

In summary, we make the following main contributions:

- We present CREB, the first comprehensive study of crash recovery bugs in large-scale distributed systems. Our in-depth analysis on four distributed systems reveals common vulnerabilities in their crash recovery processes.
- The findings in our study open up new research directions. We hope our study can not only improve existing approaches, but also shed new light on combating crash recovery bugs.
- We provide a large-scale benchmark of crash recovery bugs in distributed systems, which can be used to evaluate the effectiveness of tools in combating crash recovery bugs.

2 CRASH RECOVERY MODEL

To help understand crash recovery bugs, we build a general node crash recovery model for distributed systems by investigating crash recovery mechanisms in popular open-source distributed systems, e.g., ZooKeeper [49], Hadoop MapReduce [47], Cassandra [45] and HBase [48]. Note that these systems involve different architectures and crash recovery mechanisms (see more details in Section 3.1). For example, Hadoop MapReduce uses a master / slave architecture, and Cassandra uses a peer-to-peer architecture.

To simplify the presentation, we use a two-node distributed system in Figure 1 to demonstrate the general crash recovery model. A node in a distributed system usually contains five execution stages, shown as solid rectangles in Figure 1. Before providing

normal services (*serving stage*), e.g., computing, a node first goes into a *startup stage* to prepare its work environment. As the serving stage provides normal services, a node usually has specific mechanisms to report its liveness and monitor the availability of other relevant nodes in the cluster (*node monitoring / reporting stage*). The node monitoring / reporting stage is relatively independent and usually works in conjunction with its counterparts at other relevant nodes via direct network connection or other mediations. *Crash tolerance stage* and *reboot tolerance stage* will be triggered respectively when a crash and a reboot are detected in the node monitoring / reporting stage.

In the five execution stages, seven crash recovery components (green rounded rectangles in Figure 1) are introduced to handle potential node crashes. We describe these components as follows.

In the serving stage of *node1*, the *backing up component* persists *node1*'s important in-memory data into backups (e.g., transaction logs in ZooKeeper [22]). These backups can be persisted in local disk or remote servers in underlying storage systems, e.g., HDFS [54]. If *node1* comes back from a crash, it can recover its in-memory data according to these backups.

When *node1* crashes due to certain event (step 1), e.g., a power failure, it loses all its in-memory data and stops execution instantly. This crash can be detected by other live nodes who care about *node1*'s state, e.g., *node2*, through certain detection mechanisms in the *crash detection component*, e.g., heartbeats. Then, *node2* starts to handle the crash of *node1* (step 2). For example, the *crash handling component* may clean up stale information left by *node1* and take over *node1*'s tasks. Finally, the whole system goes back into a consistent state, without *node1*.

In some cases, *node1* will be rebooted. It first goes into the startup stage (step 3). In this stage, a *local recovery component* retrieves backups that are written by *node1* before crash, and updates *node1*'s state. The *remote synchronization component* reports *node1*'s current state to other relevant nodes (e.g., *node2*) and synchronizes with them. In the last, node monitoring / reporting stage and serving stage are activated (steps 5 and 6), and then *node1* can provide normal services.

During the startup of *node1*, some live nodes who care about *node1*'s state may detect *node1*'s reboot. For example, the *reboot detection component* in *node2* reports the reboot and starts the reboot tolerance stage (step 4). The *reboot handling component* will update the corresponding states and takes actions for the reboot. For example, *node2* may work out a recovery plan for *node1* and synchronizes *node1*'s state with other nodes, especially when *node2* is the master node.

Note that, in our general crash recovery model, we do not distinguish the architectures (e.g., master / slave in MapReduce [47] and peer-to-peer in Cassandra [45]) and crash recovery mechanisms (e.g., write-ahead logging [59] and hinted handoff [55]). Our general crash recovery model can be instantiated as different crash recovery mechanisms in different distributed systems. A node may not involve all execution stages and crash recovery components when handling a crash and the corresponding reboot. For example, if *node1* and *node2* are both slave nodes in HBase, they do not care about each other. Thus, they will not have node monitoring / reporting stages for each other. But if the master node detects the crash of *node1* and then transfers *node1*'s data to *node2*, the crash handling component in *node2* is triggered by the master node to take over *node1*'s data. If *node1* and *node2* are both peer nodes in Cassandra, the node monitoring / reporting stage in *node2* will detect *node1*'s crash when it does not receive acknowledges of gossip messages sending to *node1* in the timeout period or it receives gossip messages from other nodes that mark *node1* as dead. Then *node2* starts its crash tolerance stage to update *node1* as dead in its local state.

Table 1: Target Distributed Systems and Studied Crash Recovery Bugs

System	Domain	Architecture	# of bugs
ZooKeeper	Coordination	Master/slave based on leader election	22
Hadoop MapReduce	Computing framework	Master/slave	22
Cassandra	Storage	Peer-to-peer	27
HBase	Storage	Master/slave	32
Total	-	-	**103**

3 METHODOLOGY

3.1 Target Systems

We select crash recovery bugs from four popular open-source distributed systems: ZooKeeper [49], Hadoop MapReduce [47], Cassandra [45], and HBase [48]. These systems represent different kinds of distributed systems: Hadoop MapReduce for distributed computing frameworks, Cassandra and HBase for storage systems, and ZooKeeper for coordination services. As shown in Table 1, these four systems cover two common architecture models: peer-to-peer and master / slave. To combat node crashes, these systems contain various sophisticated crash recovery mechanisms. In the following, we briefly introduce the four target systems and their crash recovery mechanisms.

ZooKeeper (ZK for short) is a centralized coordination service for distributed applications. A Zookeeper cluster has a master node (i.e., leader) and several slave nodes (i.e., followers). ZooKeeper adopts an agreement protocol to handle write requests. When the leader crashes, all followers will restart the leader election. The election ends with a new leader if it has a majority of supporters. Each node in a ZooKeeper cluster maintains some in-memory states, along with transaction logs and snapshots in disk. Writes from clients are first serialized to log files before being applied to in-memory states. Periodically, a node takes a snapshot of its in-memory states, to ease crash recovery.

Hadoop MapReduce (MR for short) is a distributed computation framework for processing large data in a reliable manner. The second-generation MapReduce (MRv2) adopts YARN [26] to separates resource management from job scheduling. In MRv2, there is a ResourceManager (RM), a NodeManager (NM) per server, and an AppMaster (AM) per application. MRv2 restarts an AM for a failed job or reschedule a failed task. Most crash recovery bugs in MapReduce in our study belong to MRv2.

HBase (HB for short) is a distributed big data storage system with a master / slave architecture. HBase uses HDFS to store its data files, e.g., write-ahead logging files (i.e., HLog), etc. HBase[1] uses ZooKeeper to monitor its nodes and store some metadata, e.g., region transition state. In an HBase cluster, the master node HMaster (HM) is responsible for monitoring all RegionServers,

[1] Our studied HBase bugs belong to HBase 0.90 up to 1.0. In these versions, HBase uses ZooKeeper to maintain state consistence among nodes.

and all metadata must be changed through HMaster. A RegionServer (RS) is responsible for managing data and providing services for clients. When the HM crashes, one of backup master nodes will be activated. When a RS crashes, the HM detects the RS crash, and reassigns its regions to other RSs.

Cassandra (CA for short) is a scalable database based on a peer-to-peer architecture. Each node in a Cassandra cluster plays the same role. Cassandra adopts the Gossip protocol to communicate among nodes. In Cassandra, each piece of data has more than one (e.g., 3) replica. A write operation is first serialized to the commit log before being written to the in-memory data structure memtable for crash recovery. When the memtable is full, it will be flushed into the SSTable data file. If a node crashes, updates to the dead node can be kept as hinted handoffs in a coordinator node. When the dead node comes back, updates in hinted handoffs will be applied to the node to carry out the recovery process.

3.2 Collecting Crash Recovery Bugs

Our four target distributed systems all have well-organized and publicly available issue repositories (i.e., JIRA). So far, there are about 42,000 issues in these systems. Investigating all these issues is a daunting and time-consuming task. Instead, we start our study from an existing cloud bug study database, CBS [15]. CBS contains 3,655 vital issues from six distributed systems (ZooKeeper [49], Hadoop MapReduce [47], Cassandra [45], HBase [48], HDFS [54] and Flume [46]), reported from January 2011 to January 2014. We select CBS as our study base due to the following two reasons. First, CBS has covered all our target distributed systems. Second, CBS provides some labels that can help us classify and understand bugs.

Unfortunately, we cannot find out all crash recovery bugs in CBS by simply searching labels such as "crash" and "reboot". Thus, we select crash recovery bugs via the following steps. (1) We read the description in each issue report and keep the issue once it satisfies the following conditions: available fixes, clear description, and related to crash recovery. To decide if an issue is related to crash recovery, we look for relevant keywords in the bug reports, e.g., "crash", "kill", "fail", "stop", "shutdown", "leave", "reboot", "restart", "failover", "recovery", and system-specific keywords, e.g., "ServerShutdownHandler" in HBase and "hinted handoffs" in Cassandra. In some issue reports, developers have provided bug scenarios, which can help us determine if they are related to crash recovery. (2) We carefully read the issue description, developer comments and patches, and rebuild bug scenarios step by step. We keep an issue as a crash recovery bug if it needs at least one crash to trigger. If a bug has to be triggered through a "stop" command rather than a crash, which means the bug is caused by faults in the stopping process, then we do not take the bug as a crash recovery bug. For example, in bug c2072,[2] a client sends a decommission command to a node to remove it from the cluster. The node cannot be marked as "left" due to a concurrency issue in the leaving process. In this case, we cannot use a crash to replace the decommission command. So, it is not considered as a crash recovery bug. Furthermore, we only keep crash recovery bugs that we can completely understand. As shown in Table 1, we collect 103 crash recovery bugs, including 22 from ZooKeeper, 22 from Hadoop MapReduce, 27 from Cassandra and 32 from HBase.

[2] "c" denotes Cassandra. 2072 is bug ID. All bugs in the paper are represented in this form. "m" denotes Hadoop MapReduce. "h" denotes HBase. "c" denotes Cassandra. All bug examples in this paper have accessible links.

3.3 Analyzing Crash Recovery Bugs

To answer our four research questions, we perform an in-depth analysis of the 103 crash recovery bugs, according to their details, e.g., comments, patches and source code. In this process, we write down detailed bug scenarios step by step for each bug. We further assign them into different categories according to root causes, bug manifestation, bug impacts and bug fixing.

Note that a crash recovery bug usually causes system failures after a long propagation chain. So, we only categorize its root cause according to its initial fault. For example, under an unexpected crash, a non-atomic update to backups may leave some corrupted files, and then the recovery process cannot handle these corrupted files and throws exceptions. We categorize this bug into incorrect backing up although the incorrect handling of corrupted files in the recovery process triggers system failures. For bug manifestation and bug impacts, we refer to related work [15][28] for initial categorization, e.g., the number of crashes and reboots, and performance degradation. In this process, we also create new categories for bugs that do not belong to any known categories, delete useless categories, and refine categories.

Through analyzing these 103 crash recovery bugs, we obtain many interesting findings (Section 4-7). Based on these findings, we further summarize lessons learned and implications to existing approaches in combating crash recovery bugs in Section 8.

3.4 Threats to Validity

We only select bugs from CBS [15] in four open-source distributed systems. They do not cover crash recovery bugs in all kinds of distributed systems, and new bugs submitted after January 2014 are not included. Nonetheless, these four systems are widely used and cover a diverse set of architectures. Their mechanisms for combating crashes, e.g., write-ahead logging, hinted handoffs and replicas, are also commonly used by other systems. Therefore, our studied bugs are representative.

For each bug, we carefully study its description, patches and discussions among developers, and read source code to have a deep understanding. We then write down all steps to reproduce the bug. Finally, we make sure that crashes or restarts are an essential condition for the bug. All studied bugs have been discussed and confirmed by at least two authors in this paper. Thus, we believe that all studied bugs are true positives and have been thoroughly studied. Note that we exclude bugs without clear descriptions and fixes to maintain the accuracy of our study results.

4 ROOT CAUSE

4.1 Bug Pattern

We classify our studied crash recovery bugs into five bug patterns according to their root causes as shown in Table 2.

4.1.1 Incorrect Backup

Finding 1: *In 17/103 crash recovery bugs, in-memory data are not backed up, or backups are not properly managed.*

When a node crashes, all its in-memory data are lost. Thus, a node needs to back up its important in-memory data to facilitate crash recovery. Backups can be stored in local disk, or a distributed file system, such as HDFS [54]. When a crash node is rebooted, it

Table 2: Bug Patterns

Bug patterns		ZK	MR	CA	HB	Total
Backup	No backup	4	2	0	0	6
	Incorrect backup management	3	3	4	1	11
Crash/reboot detection	No crash detection	0	5	2	7	14
	Untimely crash/reboot detection	0	1	2	1	4
Incorrect state identification		5	4	5	3	17
Incorrect state recovery		5	4	11	9	29
Concurrency		5	3	3	11	22

may restore its data from its backups. Developers need to consider what data should be backed up, how to perform backups, and where to store backups. Faults in the above processes can make crash recovery fail. We further classify them into two categories.

No backup (6 bugs). In-memory data that are necessary for crash recovery should be backed up. Forgetting to back up important data can cause bad consequences when suffering from node crashes. Take bug z1264 in Figure 2 as an example. In ZooKeeper, every transaction should be persisted in the transaction log before being applied in memory. In the startup stage, the follower performs synchronization with the leader. During this process, a commit message *TXN_A* arrives in the follower side and is applied directly in memory without being persisted in the transaction log due to the incorrect design of the protocol. In the serving stage, another transaction *TXN_B* arrives, and is persisted correctly. If the follower crashes after *TXN_B*, and then reboots, it cannot recover *TXN_A*, since it considers all transactions before *TXN_B* have been correctly recovered through backups. Therefore, *TXN_A* is lost in the follower.

Incorrect backup management (11 bugs). Backups should be updated and managed correctly. Otherwise, the backup data become incorrect, and distributed systems cannot recover from these incorrect backups. There are two bug scenarios here.

- Backups are not updated correctly. (1) The update logic is incorrectly implemented. For example, in bug z1489, appending a transaction into the transaction log file after a truncate command without reopening the log file leaves a "hole" in the log file. 2 bugs belong to this case. (2) Backups should be updated in an atomic way. However, a crash can break this atomicity. For example, in bug z1427, two update operations on a backup file is broken by a crash, and the backup file is corrupted [34][44]. In bug z1653, two backup files are expected to be updated atomically. Unfortunately, a crash may happen in between. In this case, the first file is updated, and the second is not updated timely. 6 bugs belong to this case.
- To save disk space, stale backups need to be cleaned up. However, premature removal of backups can cause crash recovery to fail. 3 bugs belong to this case. For example, in bug m5476, after a job is finished, the AM first deletes its staging directory and then unregisters itself with the RM. If a crash occurs in between, then the RM restarts a new AM. The new AM fails since the staging directory has already been removed.

4.1.2 Incorrect Crash/Reboot Detection

Finding 2: *In 18/103 crash recovery bugs, crashes and reboots are not detected or not timely detected.*

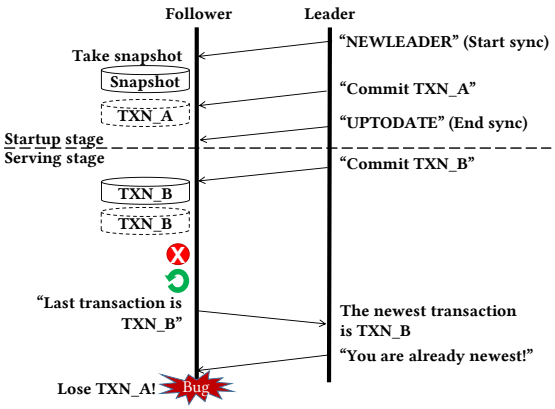

Figure 2: Bug z1264. *The dashed cylinders denote data is stored in memory, and solid ones denotes data is stored in files.*

When a node crash / reboot occurs, other relevant live nodes should detect the crash / reboot through certain mechanisms, e.g., timeout, heartbeat, or Gossip. If a node crash / reboot is not detected in a timely manner, the corresponding crash recovery will not be applied. 18 bugs are caused by incorrect crash / reboot detection. We further classify them into two sub-categories.

No crash detection (14 bugs). Although our four studied distributed systems all have dedicated mechanisms to detect crashes, some crashes are not detected in the following three cases.

- When a node crashes, some other relevant nodes may access the crash node without perceiving that the node has crashed. These relevant nodes may hang or throw errors. 9 bugs belong to this case. In bug m3228, a MapReduce job is executing some tasks in containers managed by a NM. Then, the server that holds the NM and tasks crashes. After a while, the AM marks these tasks as timeout since it does not receive heartbeats from tasks in a given time and notifies the NM to stop the containers in which these tasks run. However, the AM keeps waiting for the response from the NM forever since the NM is down.
- If a crash node reboots very quickly, then the crash may be overlooked by the crash detection component based on timeout. Thus, the crash node may contain corrupted states, and no one knows it. 2 bugs belong to this case. In bug m3186, the RM loses all its metadata about jobs after a quick reboot. However, the AM does not know the crash and the reboot, and still commits jobs to the RM, resulting failures.
- Crashes have been detected, however, no crash recovery can be applied. We classify this case into no crash detection, since their induced results are the same. 3 bugs belong to this case. In bug h5918, if the HM crashes and reboots, it will try to assign the *ROOT* region and the *META* region[3] in the startup stage. The *ROOT* region is assigned to *RS1* first. Then the HM tries to assign the *META* region. This assignment needs to access the *ROOT* region. If *RS1* crashes at this time, the *ROOT* region will become unavailable and cannot be recovered since the crash recovery handler *ServerShutdownHandler* is not enabled yet. Thus, the assignment of *META* region will wait for an available *ROOT* region forever.

Untimely crash / reboot detection (4 bugs). A crash / reboot should be detected as soon as possible. If it takes a very long time

[3] ROOT tracks META region; META region stores information about data regions.

to confirm a crash, the crash recovery will be delayed. For example, in bug c3273, nodes in Cassandra use the accrual failure detector [20] to detect node crashes. The timeout threshold is decided dynamically and related to history time intervals of message arrival. In corner scenarios, the timeout threshold can become large (e.g., 10 minutes), and a node crash cannot be detected timely. Another scenario is once a node is rebooted, distributed systems should make it available to serve as soon as possible. In bug h4397, all RSs are down, and thus all region assignments cannot be processed. So, the *TimeoutMonitor* takes over, and checks if there are available RSs after 30 minutes. Some RSs may come back in less than 30 minutes. However, the *TimeoutMonitor* still waits for at least 30 minutes, and no region assignment can be processed.

4.1.3 Incorrect State Identification

Finding 3: *In 17/103 crash recovery bugs, the states after crashes / reboots are incorrectly identified.*

When a node crashes unexpectedly, it may leave some stale information, e.g., unfinished tasks and stale states. These leftovers (i.e., stale information) can be persisted files, in-memory data or threads / processes in other live nodes. A crash recovery process should precisely identify these leftovers after a crash happens. When a node reboots, it should correctly identify its final state and the state of current cluster. Meanwhile, other relevant live nodes should correctly identify the state of the reboot node. Incorrect state identification can make the crash recovery process adopt incorrect recovery decisions. There are three cases in this category.

- The recovery process misses the correct states (3 bugs). For example, in bug z582, a new leader needs to restore the most recent state from backups, i.e., transaction logs and snapshots. However, it only considers transaction logs. If the transaction logs are deleted, the state cannot be restored correctly.
- The recovery process mistakenly considers wrong states as correct (12 bugs). For example, in bug z975, *node1* crashes after it is voted as the leader. When *node1* is rebooted, it may receive stale vote messages for *node1*. Then, *node1* mistakenly takes these stale messages and sets itself as the leader.
- Two bugs are caused by incorrect data parser logic. In bug c4842, the recovery process cannot retrieve a column family containing metadata *DateType* in *CompositeType*, since it cannot separate *DateType* from *CompositeType* correctly.

4.1.4 Incorrect State Recovery

Finding 4: *The states after crashes / reboots are incorrectly recovered in 29/103 crash recovery bugs. Among them, 14 bugs are caused by no handling or untimely handling of certain leftovers.*

When a crash or a reboot happens, four recovery components in Figure 1, i.e., local recovery, remote synchronization, crash handling and reboot handling, will try to recover the cluster to a consistent state. Wrong state recovery will introduce crash recovery bugs. 29 bugs belong to this case.

We are more interested in the leftovers of a crash node. When a node crashes suddenly, the information maintained by the crash node may need to be transferred, and the states left in the cluster may be corrupted, etc. These leftovers should be correctly handled, e.g., deleted or updated. Among 29 bugs caused by incorrect state recovery, 15 bugs are caused by incorrectly handling of leftovers. Among them, we find two interesting patterns, covering 14 bugs.

- No handling of leftovers (11 bugs). For example, in bug m3858, a task node crashes in the commit phase, and leaves a *taskID* in the commit attempt list in the AM. When recovering from the crash, the AM starts another task node to complete the task without deleting this *taskID*. When the new task node attempts to commit the task, it fails since the *taskID* already exists in the commit list.
- Untimely handling of leftovers (3 bugs). For example, in bug h6060, a RS is opening a region R1, and a crash happens to the RS while R1 is still in opening state. The crash recovery component *ServerShutdownHandler* does not reassign R1 to another live RS immediately, but leaves it to be reassigned by the *TimeoutMonitor* after 30 minutes.

4.1.5 Concurrency

Finding 5: *The concurrency caused by crash recovery processes is responsible for 22/103 crash recovery bugs.*

Distributed systems execute many protocols concurrently on thousands of nodes with no common clocks and may trigger various concurrency issues. When a node crashes / reboots, the recovery process is usually concurrent with other normal processes. Concurrency bugs can manifest in these concurrent processes. 22 bugs belong to this category. We further classify them into 3 subcategories according to their involved processes.

- Concurrency in one recovery process (5 bugs). Take bug c2083 as an example. In Cassandra, after *node1* crashes and then reboots, its coordinator node *node2* will recover *node1*'s data through hinted handoffs. During *node1*'s downtime, a column family is created, and new data are inserted into the column family. After *node1* reboots, the column family creation message and the new data are sent to *node1* concurrently. Thus, if the new data arrive at *node1* earlier than the column family creation message, then a failure will happen, since the column family is not created yet.
- Concurrency between two recovery processes (4 bugs). For example, in bug h5179, a standby HM is activated since the primary HM is down. The standby HM first performs a startup process. During this process, a RS crashes. The crash handling component *ServerShutdownHandler* starts handling the RS crash. At the same time, the startup stage in HM finds that the RS is dead and also tries to handle it, and thus the double assignment for the RS is triggered.
- Concurrency between a recovery process and a normal execution (13 bugs). For example, in bug h9514, a RS crashes and the HM handles the crash using *ServerShutdownHandler*. This handler first splits the log for the dead RS and then reassigns regions on the RS to other RSs. When a client request that wants to assign a region on the dead RS arrives concurrently and is executed before splitting the log in *ServerShutdownHandler*, some data could be lost.

4.2 Components of Root Causes

Finding 6: *All the seven recovery components in our crash recovery model can be incorrect and introduce bugs. About one third of bugs are caused in crash handling component.*

As shown in Section 2, we separate the crash recovery process into seven crash recovery components. We wonder whether crash recovery bugs can be easily introduced in some components. Thus,

Table 3: Statistics of Involved Crash Recovery Components

Components	ZK	MR	CA	HB	Total
Crash handling	2	6	3	24	35
Backing up	10	5	4	1	20
Local recovery	4	5	7	1	17
Crash detection	0	6	3	4	13
Remote synchronization	3	0	3	4	10
Reboot handling	3	1	6	0	10
Reboot detection	0	0	1	1	2

we put the 103 bugs into their corresponding crash recovery components according to their root causes. The result is shown in Table 3. Note that, the total number exceeds 103 since some bugs involve more than one recovery component.

As shown in Table 3, all the seven crash recovery components can be incorrect and introduce bugs. Overall, the crash handling component is the most error-prone (34%). The next top three components, backing up (19%), local recovery (17%) and crash detection (13%) also occupy a large portion. But all four systems have few bugs in reboot detection components. For ZooKeeper, about a half of the crash recovery bugs are rooted in the backing up component. For MapReduce, we do not find bugs in the remote synchronization component. For HBase, over two thirds of its bugs are caused by the crash handling component.

4.3 Crashes on/during Recovery

Finding 7: *In 15/103 crash recovery bugs, new relevant crashes occur on / during the crash recovery process, and thus trigger failures.*

Crashes can happen at any time. New node crashes can happen when a node crash / reboot is being handled, i.e., crash handling, reboot handling, local recovery and remote synchronization in Figure 1. The new crashes can complicate the current recovery process, and cause failures. 15 bugs fall into this category.

- The node executing the recovery process crashes, and the crash recovery process is interrupted (7 bugs). For example, in bug h3596, *RS1* first crashes. *RS2* tries to take over *RS1*'s regions and sets a lock znode in ZooKeeper. Now, *RS2* crashes before releasing the lock. Regions in *RS1* will be locked forever.
- During the recovery process of a node, another node crashes (8 bugs). This mainly involves two kinds of scenarios: (1) Current crash recovery process may need to access other nodes, but the accessed nodes crash. Then, the crash recovery process hangs or fails. For example, in bug h3446, the *META-RS* crashes when handling a normal RS's crash. Then the recovery process for the normal RS encounters a server-not-running exception and orphans lots of regions. (2) The two crash recovery processes may access share data and introduce bugs. For example, in bug h5179 we have discussed in Section 4.1.5, during the startup stage of HM, a RS crashes. The RS crash is handled twice, and thus causes data loss.

5 TRIGGERING CONDITIONS

To trigger a crash recovery bug, only crashes and reboots may not be sufficient. We should consider other input conditions. We measure the complexity of input conditions (Section 5.1) to trigger a bug, e.g., the number of nodes and special configurations. We further discuss the timing about crashes and reboots (Section 5.2).

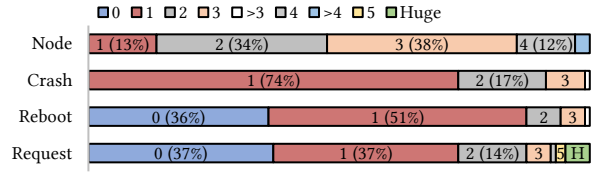

Figure 3: Basic input conditions. *It shows the minimum numbers of nodes, crashes, reboots and client requests required by the 103 crash recovery bugs. H means a large number, e.g., 1000.*

5.1 Input Conditions

Finding 8: *Almost all (97%) crash recovery bugs involve four nodes or fewer.*

Finding 9: *No more than three crashes can trigger almost all (99%) crash recovery bugs. No more than one reboot can trigger 87% of the bugs. In total, a combination of no more than three crashes and no more than one reboot can trigger 87% (90 out of 103) of the bugs.*

Finding 10: *63% of crash recovery bugs require at least one client request, but 92% of the bugs require no more than 3 user requests.*

First, we count the basic input conditions of crash recovery bugs, i.e., the minimum numbers of nodes, crashes, reboots and user requests involved in triggering a bug. As show in Figure 3, we find that 85% (88 out of 103) of bugs involve no more than three nodes. 74% (76 out of 103) of bugs can be reproduced by only injecting one crash. Almost all bugs (99%) can be triggered by no more than three crashes. 36% of bugs do not need to inject any reboot, and 51% bugs only need one reboot. 87% (90 out of 103) of bugs can be triggered with a combination of no more than three crashes and no more than one reboot. For client requests, 37% (38 out of 103) of bugs do not need client requests. 37% of bugs only need one client request. That means crash recovery bugs can be triggered by simple client inputs. Note that there are five bugs that need large number of client requests, e.g., thousands.

Finding 11: *38% of crash recovery bugs require complicated input conditions, e.g., special configurations or background services.*

Besides the basic conditions as shown above, 39 bugs require complicated input conditions. We summarize these complicated input conditions in Figure 4, and briefly explain them as follows.

Special configurations. 18 crash recovery bugs only manifest themselves under special configurations. For example, bug c2083 can only be manifested when the hinted handoff is enabled.

Background services. 11 bugs need background services, e.g., load balancing, or compaction of persistent files, and snapshotting. These background services usually happen non-deterministically. For example, bug z1573 requires that the crash node is snapshotting its in-memory state. The snapshotting process rarely happens since it occurs only when the transaction log file is full.

There are other types of triggering conditions. For example, 5 bugs need man-made corrupted data, 4 bugs need one or several nodes to start from scratch, and 2 bugs need a network partition.

5.2 Crash / Reboot Triggering Window

Finding 12: *The timing of crashes / reboots is important for reproducing crash recovery bugs.*

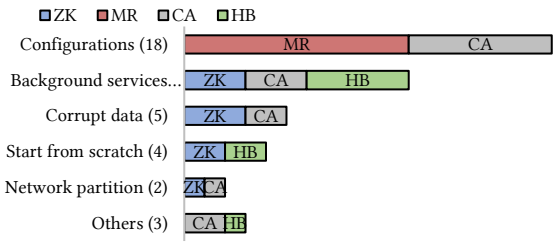

Figure 4: Special input conditions. *It shows the numbers (in the parentheses) of crash recovery bugs that require specific configuration, background services, corrupt data, start from scratch, network partition, and other conditions, respectively.*

Only node crashes that happen at certain system states can trigger crash recovery bugs. Accordingly, the difficulty levels of triggering crash recovery bugs are different. We use crash / reboot windows to measure such difficulty levels, as shown in Table 4.

We divide the crash triggering window into 3 categories, according to the follow rules.

- Easy (32 bugs). To trigger a crash recovery bug, the crash can occur at a consistent global state, e.g., the ZooKeeper cluster finishes the leader election, or a client request has been processed. In this case, we do not need to control the internal states of some nodes and can easily inject crashes when the cluster stays in a consistent state.
- Moderate (53 bugs). The node crash needs to occur when the node stays in certain states. That is, the timing of the crash only relates to the events happening in the same node. Thus, we only need to consider the internal states in one node.
- Difficult (18 bugs). The node crash needs to occur when other nodes are in certain states. Thus, we need to inject the crash after checking the states of other nodes.

While reproducing each bug requires at least one crash, only 66 bugs need reboots. We divide reboot triggering windows into 2 categories, according to the follow rules. Note that, we do not have a moderate level for reboot triggering windows since a dead node does not have a chance to change its states.

- Easy (57 bugs). The reboot can happen at any time or after a fixed amount of time. For example, in bug c2115, the reboot should happen after the node stays down for a fixed time. We can easily control this downtime.
- Difficult (9 bugs). The reboot needs to occur when other nodes stay in certain states. Thus, we need to inject the reboot after checking the states of other nodes.

Note that our decision about how difficult to inject a crash / reboot is based on the above rules. We believe our results can indicate the difficulty of reproducing crash recovery bugs.

6 BUG IMPACTS

Finding 13: *Crash recovery bugs always have severe impacts on reliability and availability of distributed systems. 38% of the bugs can cause node downtimes, including cluster out of service and unavailable nodes.*

To better understand how severe a crash recovery bug is, we study the failure symptom of each bug. The crash recovery bugs influence availability, reliability, and performance of the target systems. We classify them into five categories.

Table 4: Crash / Reboot Triggering Windows

Triggering window		ZK	MR	CA	HB	Total
Crash window	Easy	10	2	17	3	32
	Moderate	12	16	9	16	53
	Difficult	0	4	1	13	18
Reboot window	Easy	21	5	21	10	57
	Difficult	0	3	3	3	9

Cluster out of service. 11 bugs (11%) take down the whole cluster. Bugs in the dominant nodes (e.g., master) are more likely to cause a cluster out of service. For example, bug z1419 causes endless leader election, and the whole cluster cannot serve clients.

Unavailable nodes. 28 bugs (27%) cause unavailable nodes, e.g., node hangs and missing a live node. However, the cluster can still provide services.

Data related failures. 22 bugs (21%) cause data related failures such as data loss, data inconsistency, and data staleness.

Performance degradation. 13 bugs (13%) cause performance degradation, e.g., much more time for crash recovery.

Operation failures. 29 bugs (28%) cause operation failures, e.g., failures or hangs in an operation. We only put a bug into operation failures when it does not cause other failures above to avoid double counting.

Note that, 38% of crash recovery bugs can cause downtimes of the cluster or some nodes. Compared to the bugs in CBS [15] (18%) and TaxDC [28] (17%), crash recovery bugs are more likely to cause fatal failures.

7 BUG FIXING

In this section, we discuss how developers fix crash recovery bugs, whether the fixes are complete, and how difficult to fix them.

7.1 Fix Strategies

Fixes of crash recovery bugs are highly related to their root causes and system-specific. We have not extracted clear fix patterns in our study. Thus, we only briefly discuss some fixing examples.

For bugs caused by no crash detection, a simple fix is adding a timer. For example, bug m3228 discussed in Section 4.1.2 was fixed by adding a timer when waiting for the NM to stop containers. Bugs caused by non-atomic updates to backups can be fixed by using a temporary file, e.g., bug z1427. But, the similar bug z1653 we discussed in Section 4.1.1 was fixed by detecting the unexpected crash between file updates and fixing inconsistency in two files. The fix of bug z1653 creates a special file before starting to write files, and then deletes the file after all files have been updated. If a crash happens between two file updates, the special file will be left in disk. Thus, after reboot, the system will know the crash and rewrite the unfinished file rather than stopping itself.

7.2 Fix Completeness

Finding 14: *Crash recovery bugs are difficult to fix. 12% of the fixes are incomplete, and 6% of the fixes only reduce the possibility of bug occurrence.*

Fixes of crash recovery bugs from developers may not be perfect. As shown in Table 5, we find that the fixes of 12 crash recovery bugs are incomplete. After applying the fixes, crash recovery bugs can still occur.

Table 5: Fix Completeness

	Complete fixes	Incomplete fixes		
		Omit cases	Introduce new bugs	Reduce probability
ZK	17	2	2	1
MR	19	0	2	1
CA	24	0	0	3
HB	31	0	0	1
Total	91	2	4	6

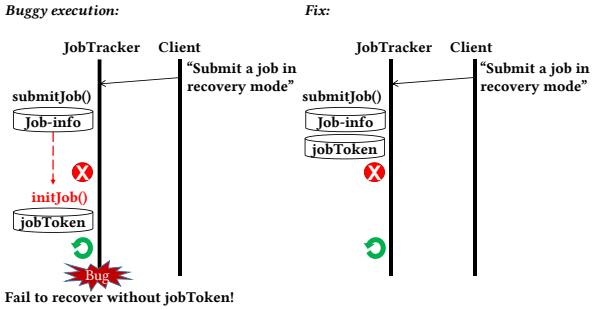

Figure 5: Incomplete fix for bug m5169. *The left part shows the buggy execution. The right part shows the fix.*

Omit some cases. The fixes of 2 bugs omit to consider some new crash scenarios. For example, in the fix of bug z1264 in Figure 2, developers add backups for update transactions, but forget to consider new session transactions committed during the synchronization process. Thus, this fix cannot avoid bug z1367.

Introduce new bugs. The fixes of 4 bugs introduce new bugs. For example, the fix of bug z596 makes a leader to load data twice during startup process. But developers forget to clean up all data between these two loads. Thus, the fix introduces a new bug z1448.

Reduce probability of bug occurrence. The fixes of 6 bugs only reduce the chance of bug occurrence, instead of eliminating the bugs. Figure 5 shows an example, bug m5169. The JobTracker[4] crashes after it updates the *job-info* file and before it stores the *jobToken*. After reboot, it tries to recover the job according to the *job-info* file but fails because it cannot find the *jobToken*. This bug is fixed by generating the *jobToken* along with the *job-info* file to reduce the chance of unexpected crashes between them.

7.3 Fix Complexity

Finding 15: *Crash recovery bug fixing is complicated. Amounts of developer efforts were spent on these fixes.*

We measure the fix complexity using five metrics: (1) the number of discussion comments; (2) the number of watchers; (3) the number of days to fix bugs; (4) the number of patches submitted; and (5) the lines of code changed in the final patch.

We show the results in Table 6. On average, crash recovery bugs involve 26 discussion comments, 5 watchers, 92 days to fix, 4 patches submitted, and 117 lines of code change.

8 LESSONS LEARNED

In this section, we discuss lessons learned, implications to existing approaches and opportunities for new research in combating crash recovery bugs in distributed systems.

[4] It is a concept from MRv1. It can be seen as a combination of AM and RM.

Table 6: Fix Complexity

	25th percentile	Median	75th percentile	Max
# of comments	11	19	29	168
# of watchers	3	4	7	28
Time (# of days)	5	13	61	1121
# of patches	1	3	5	24
LOC of final patch	10	34	95	1839

8.1 Crash Recovery Bug Detection

Crash recovery bugs in distributed systems can cause severe consequences (Finding 13), and thus resolving these bugs is of great significance for the reliability of distributed systems. Existing crash recovery bug detection approaches, e.g., SAMC [27], PACE [1] and ELEVEN82 [25] mainly focus on model checking and testing, and do not understand bug patterns behind these bugs. Our study reveals that crash recovery bugs can be summarized into 5 kinds of bug patterns, i.e., incorrect backup (Finding 1), incorrect crash / reboot detection (Finding 2), incorrect state identification (Finding 3), incorrect state recovery (Finding 4) and concurrency (Finding 5). These bug patterns shed new light and guidance on crash recovery bug detection based on program analysis.

Backup-guided bug detection. To recover from crashes, important in-memory data should be persisted. Finding 1 implies that no backup of such data can introduce crash recovery bugs. This suggests that it is possible to detect bugs by analyzing the backing up component, without injecting crashes / reboots. The key challenge is to understand what in-memory data should be backed up. We find that some in-memory data are backed up in some program paths and not in other program paths. This indicates that the data may be backed up incorrectly (e.g., bug z1264). From another perspective, developers can annotate what data should be backed up, and then we use the annotations to analyze whether they are backed up properly. Although this requires some efforts from developers, it will greatly help bug detection.

Crash / reboot detection analysis. Crashes and reboots can happen at any time in any node. However, if crashes / reboots are not detected properly, no crash recovery mechanisms will be applied. Finding 2 indicates that many crash recovery bugs are caused by incorrect crash / reboot detection. Our study shows that a crash / reboot at any time should be detected. We can use these patterns to detect bugs with incorrect crash / reboot detection.

State inconsistency guided detection. To recover from a crash or reboot, correct states must be identified first. Finding 3 indicates that states can be incorrectly identified, leading to crash recovery bugs. We can compare the states before crashes / reboots with the states in crash recovery, to identify inconsistencies. These inconsistencies may indicate crash recovery bugs. Finding 4 indicates states can be incorrectly updated during crash recovery, e.g., leftovers of a crash node are not cleaned up / updated. This pattern can be used to detect incorrect state recovery bugs.

Concurrency analysis in crash recovery bugs. Finding 5 implies that crash recovery processes can introduce more concurrency into distributed systems, and thus trigger more complicated concurrency bugs. We notice that when the recovery process concurrently accesses the same resources (e.g., a region in HBase), it can easily trigger bugs. However, existing concurrency detection tools cannot handle crashes / reboots, e.g., DCatch [30]. Further

studies can be performed to handle concurrency during crash recovery. We also need to consider more concurrency scenarios, e.g., concurrency among two crash recovery processes.

Bug fix oriented detection. Fixing crash recovery bugs is complicated (Finding 15). Finding 14 implies that fixes for crash recovery bugs have a fairly high probability of being incorrect. Some fixes only reduce the possibility of bug occurrences. This shows a unique opportunity for developing new techniques to detect bugs in fixes and verify fixes for crash recovery bugs.

8.2 Testing of Distributed Systems

Software testing is crucial in exposing bugs for complex software systems. Few techniques [14][36] have been proposed for distributed system testing. Our study reveals some implications for future studies in distributed system testing.

In our study, we find that crash recovery mechanisms can contain simple errors, e.g., the backup data parse error in bug c4842. This indicates that crash recovery implementations may not be sufficiently tested. Finding 6 indicates that system testing should cover all 7 crash recovery components. Finding 8 implies that it is not necessary to utilize a large cluster to expose crash recovery bugs. Instead, a small cluster (e.g., 4 nodes) can reveal most crash recovery bugs. Finding 9 implies that we can test distributed systems via a small number of injected crashes and reboots. Finding 10 and Finding 11 imply that test input design needs to consider user requests, configurations, and so on.

8.3 Crash / Reboot Injection Strategy

Finding 12 implies that crash/reboot timing is important for exposing crash recovery bugs. Thus, randomly exploring possible crash scenarios [24][40][50][52][53] should not work well, since most of the randomly explored scenarios may not prone to trigger bugs. Additionally, Finding 9 implies that we only need to inject a limited number (e.g., 3) of crashes and reboots. Thus, a proper injection strategy can detect crash recovery bugs efficiently and effectively. Our study reveals some crash injection scenarios that can easily trigger crash recovery bugs, e.g., crash in the process of writing a file or a group of I/O operations, crashing the node that users are accessing, quickly restart within a timeout setting, crashing the underlying systems, and crashing related nodes after backups are deleted. Finding 7 indicates that crashes on / during crash recovery can easily trigger bugs, crash / reboot injection should also consider this scenario. Future tools should further explore more crash inject scenarios in our study and apply these scenarios to speed up crash recovery bug manifestation.

9 RELATED WORK

Empirical studies in distributed systems. Guo et al. [18] classified several crash recovery misbehaviors, and concluded that failure recovery must be engineered explicitly according to a "do no harm" requirement. Gunawi et al. presented CBS [15] that contains over 3,000 vital issues in six distributed systems. The crash recovery bugs we studied are selected from CBS. We further perform in-depth analysis on these bugs, which is not done by CBS. Leesatapornwongsa et al. [28] studied 104 distributed concurrency bugs from four distributed systems. Unlike our study, their work focuses on concurrency bugs, and does not try to understand why and how crashes / reboots can trigger bugs. Dai et al. [7] analyzed 156 real-world timeout issues from 11 cloud systems. Yuan et al. [42] looked deeply into 198 production failures in distributed systems to understand how one or multiple faults eventually evolve into a user-visible failure. Wang et al. [37] studied 290,000 hardware failure reports in hundreds of thousands of servers.

Bug detection in distributed systems. Some approaches have been developed to combat bugs in distributed systems. MODIST [40] is a distributed model checker that can systematically check system behaviors under all kinds of actions. SAMC [27] detects intricate bugs in distributed systems based on semantic-aware model checking. D3S [32] detects bugs by checking predicates specified by developers at runtime. Xu et al. [39] automatically finds out problems at runtime by mining console logs. PACE [1] identifies correlated crash vulnerabilities in distributed systems. DCatch [30] adopts dynamic analysis to detect concurrency bugs in distributed systems. CloudRaid [33] detects concurrency bugs in distributed systems by flipping the order of a pair of concurrent messages that always happen in a fixed order. FCatch [31] predicts time-of-fault bugs by observing possible conflicting operations under crashes. Our study sheds new light on bug detection in distributed system as discussed in Section 8.

Testing and verification. SETSUD [23] exposes system-level defects by precisely controlling the timing of perturbations with the awareness of system-internal states. PreFail [24] allows testers to write failure injection policies for space reduction. CORDS [12] tests if a distributed storage system can correctly recover from file-system faults. DEMi [35] combines DPOR [10] and delta debugging [43] to automatically minimize buggy executions. FATE and DESTINE [14] aims to exercise multiple and diverse failures by avoiding exercising the same recovery behaviors. Verdi [38], IronFleet [19] and Chapar [29] can verify distributed systems in Coq [58] and Dafny [51]. However, these formally verified systems are not totally reliable [11]. ELEVEN82 [25] tries to prove storage systems to be crash recoverable by reducing crash recoverability to reachability, in case of one crash.

10 CONCLUSIONS

Although automated recovery has been treated as a first-class operation of distributed systems, crash recovery is still complicated and error-prone. This paper presents CREB, an in-depth study of 103 crash recovery bugs in four popular open-source distributed systems. Our study reveals a number of interesting findings and implications. We hope our study can inspire more researchers to combat crash recovery bugs in distributed systems. The CREB dataset is available at http://www.tcse.cn/~wsdou/project/CREB.

ACKNOWLEDGMENTS

We thank Kang Yin, Jie Wang, Hui Li and Feng Yang for their contributions in bug analysis. This work was partially supported by National Key Research and Development Program of China (2017YFB1001804), National Natural Science Foundation of China (61732019, 61702490), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), National Science Foundation (CNS-1513120), Youth Innovation Promotion Association at Chinese Academy of Sciences, CAS/SAFEA International Partnership Program for Creative Research Teams, and Alibaba Group through Alibaba Innovative Research (AIR) Program.

REFERENCES

[1] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 151–167.

[2] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 331–346.

[3] Mike Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. 335–350.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems* 26, 2 (2008), 1–26.

[5] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15* (2015), 18–37.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*. 143–154.

[7] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. 2018. Understanding Real-World Timeout Problems in Cloud Server Systems. In *Proceeding of the IEEE International Conference on Cloud Engineering (IC2E)*. 1–11.

[8] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI)*. 137–149.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP)*. 205–220.

[10] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 110–121.

[11] Pedro Fonseca. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*. 328–343.

[12] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-dusseau, and Remzi H Arpaci-dusseau. 2017. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST)*. 149–165.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*. 29–43.

[14] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 238–252.

[15] 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. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*. 1–14.

[16] Haryadi S Gunawi, Abhishek Rajimwale, Andrea C Arpaci-dusseau, and Remzi H Arpaci-dusseau. 2008. SQCK: A Declarative File System Checker. In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI)*. 131–146.

[17] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*. 265–278.

[18] Zhenyu Guo, Sean Mcdirmid, Mao Yang, Li Zhuang, Pu Zhang, and Yingwei Luo. 2013. Failure Recovery: When the Cure is Worse Than the Disease. In *Proceedings of 14th Workshop on Hot Topics in Operating Systems (HotOS)*. 1–6.

[19] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. 1–17.

[20] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. 2004. The ϕ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*. 66–78.

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 295–308.

[22] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*. 11–11.

[23] Pallavi Joshi, Malay Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. 2013. SETSUDO: Perturbation-based Testing Framework for Scalable Distributed Systems Pallavi. In *Conference on Timely Results in Operating Systems (TRIOS)*. 1–14.

[24] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. 2011. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*. 171–188.

[25] Eric Koskinen and Junfeng Yang. 2016. Reducing Crash Recoverability to Reachability. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 97–108.

[26] Vinod Kumar Vavilapalli et al. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing (SOCC)*. 1–16.

[27] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 399–414.

[28] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 517–530.

[29] Mohsen Lesani, Christian J Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-Value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 357–370.

[30] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems Cloud systems. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 677–691.

[31] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[32] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. D3S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 423–437.

[33] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. CloudRaid: Hunting Concurrency Bugs in the Cloud via Log-Mining. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. To appear.

[34] Thanumalayan Sankaranarayana Pillaic, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 433–448.

[35] Colin Scott, Aurojit Panda, Arvind Krishnamurthy, Vjekoslav Brajkovic, George Necula, and Scott Shenker. 2016. Minimizing Faulty Executions of Distributed Systems. In *Proceedings of 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 291–309.

[36] Koushik Sen and Gul Agha. 2006. Automated Systematic Testing of Open Distributed Programs. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE)*. 339–356.

[37] Guosai Wang, Wei Xu, and Lifei Zhang. 2017. What Can We Learn from Four Years of Data Center Hardware Failures ? In *Proceedings of 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 25–36.

[38] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 357–368.

[39] Wei Xu, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*. 117–132.

[40] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST:

Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation (NSDI)*. 213–228.

[41] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2004. Using Model Checking to Find Serious File System Errors. In *Proceedings ofthe Sixth Symposium on Operating Systems Design and Implementation (OSDI)*. 273–288.

[42] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 249–265.

[43] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering (TSE)* 8, 2 (2002), 183–200.

[44] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zha, and Shashank Singh. 2014. Torturing Databases for Fun and Profit. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 449–464.

[45] Apache Cassandra. Retrieved from http://cassandra.apache.org.

[46] Apache Flume Project. Retrieved from http://flume.apache.org.

[47] Apache Hadoop. Retrieved from http://hadoop.apache.org.

[48] Apache HBase. Retrieved from http://hadoop.apache.org/hbase.

[49] Apache ZooKeeper. Retrieved from http://zookeeper.apache.org.

[50] Chaos Monkey. Retrieved from https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey.

[51] Dafny is a verification-aware programming language. Retrieved from https://github.com/Microsoft/dafny.

[52] Fault Injection Framework and Development Guide. Retrieved from https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/FaultInjectFramework.html.

[53] FIT: Failure Injection Testing. Retrieved from https://medium.com/netflix-techblog/fit-failure-injection-testing-35d8e2a9bb2.

[54] HDFS Architecture. Retrieved from http://hadoop.apache.org/%0Adocs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[55] HintedHandoff. Retrieved from https://wiki.apache.org/cassandra/HintedHandoff.

[56] 2016. The 10 Biggest Cloud Outages of 2016. Retrieved from http://www.crn.com/slide-shows/cloud/300083247/the-10-biggest-cloud-outages-of-2016.htm.

[57] 2017. The 10 Biggest Cloud Outages of 2017 (So Far). Retrieved from http://www.crn.com/slide-shows/cloud/300089786/the-10-biggest-cloud-outages-of-2017-so-far.htm.

[58] The Coq Proof Assistant. Retrieved from https://coq.inria.fr/.

[59] Write Ahead Log (WAL). Retrieved from http://hbase.apache.org/book.html#wal.