Understanding Node Change Bugs for Distributed Systems

Jie Lu, Liu Chen, Lian Li*, and Xiaobing Feng
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
lujie@ict.ac.cn, liuchen17z@ict.ac.cn, lianli@ict.ac.cn, fxb@ict.ac.cn

***Abstract*—Distributed systems are the fundamental infrastructure for modern cloud applications and the reliability of these systems directly impacts service availability. Distributed systems run on clusters of nodes. When the system is running, nodes can join or leave the cluster at anytime, due to unexpected failure or system maintenance. It is essential for distributed systems to tolerate such node changes. However, it is also notoriously difficult and challenging to handle node changes right. There are widely existing node change bugs which can lead to catastrophic failures. We believe that a comprehensive study on node change bugs is necessary to better prevent and diagnose node change bugs. In this paper, we perform an extensive empirical study on node change bugs. We manually went through 6,660 bug issues of 5 representative distributed systems, where 620 issues were identified as node change bugs. We studied 120 bug examples in detail to understand the root causes, the impacts, the trigger conditions and fixing strategies of node change bugs. Our findings shed lights on new detection and diagnosis techniques for node change bugs.**

In our empirical study, we develop two useful tools, NCTrigger and NPEDetector. NCTrigger helps users to automatically reproduce a node change bug by injecting node change events based on user specification. It largely reduces the manual efforts to reproduce a bug (from 2 days to less than half a day). NPEDetector is a static analysis tool to detect null pointer exception errors. We develop this tool based on our findings that node operations often lead to null pointer exception errors, and these errors share a simple common pattern. Experimental results show that this tool can detect 60 new null pointer errors, including 7 node change bugs. 23 bugs have already been patched and fixed.

***Index Terms*—distributed system, node change, shutdown, crash, reboot**

I. Introduction

In the cloud era, more and more applications are moving from local to cloud settings. These applications need to provide 24/7 online services to users and high availability is crucial: A few minutes of service outage can easily cost a company millions of dollars [1]. Distributed systems are the fundamental building blocks of modern online applications and the reliability of these systems directly impacts service availability.

Distributed systems run on clusters of nodes. It is essential for distributed systems to be elastic: Nodes may leave or join a cluster at any time, due to unexpected failure or system maintenance. Various protocols have been implemented to tolerate such node changes. For instance, HBase [2] uses Zookeeper [3] to monitor the state of each node: If one node leaves the system, it will fork a recovery thread to move the data of the leaving node to another node.

Unfortunately, it is notoriously difficult to handle node changes right: The developers need to anticipate all possible system states and reason about concurrent execution at each state. This process is error-prone and not sufficiently tested. Improper handling of node changes may lead to catastrophic failure. On March 15th 2017, Microsoft's Azure public cloud storage service outages for eight hours, due to unexpected node failures [4]. We believe that a comprehensive study of how node changes impact system behavior is essential to better detect, diagnose and prevent *Node Change Bugs*. Previous researches have focused on studying distributed concurrency bugs [5], or techniques to detect deep errors by injecting faults or node crash events [6]–[9]. However, there is no extensive study on bugs triggered by node changes.

There are four types of common node operations:

- **Shutdown**: System administrators use system-provided scripts to *gracefully* shut down a node (or a process, the two terms are exchangeable in this paper). Consequently, a series of threads will be forked (via callback functions) to handle the aftermath (e.g closing the IO, deleting temporary file).
- **Crash**: A node crashes and leaves the cluster due to unexpected hardware or software failures. No aftermath handler will be forked.
- **Reboot**: System administrators restart a node.
- **Fresh Boot**: System administrators add a new node.

We regard all bugs triggered by the above four node operations as *Node Change Bugs*, abbreviated as NCBugs. Node operations may happen at anytime when the system is running, while bugs can be triggered only at specific times. As a result, NCBugs are difficult to reproduce, diagnose, and detect. To better understand NCBugs, we have conducted an extensive empirical study: We manually went through 6,660 bug issues of 5 representative distributed systems, obtained via a keyword-based search in their bug tracking systems. We identified 620 distinct node change bugs from the 6,660 bug issues, and inspected 120 randomly selected bugs in detail.

*Corresponding author.

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

Our empirical study tries to answer the following research questions:

- **RQ1:** What are the root causes of NCBugs?
- **RQ2:** How do NCBugs impact distributed system?
- **RQ3:** How to trigger NCBugs?
- **RQ4:** How are NCBugs fixed in practise?

In addition, we develop *NCTrigger*, a tool to automatically reproduce a given node change bug. Users need to specify the targeted system, the workload, and the bug trigger condition. NCTrigger reads the input from user specification, then automatically deploys the system, runs the workload, and injects node change events to trigger the bug. The tool is very useful for reproducing and diagnosing NCBugs. With NCTrigger, we have reproduced 30 NCBugs to get a deeper understanding of their impacts. Our findings are summarized as follows:

- Most NCbugs are triggered by reboot (68.2%), crash (18.7%), or shutdown operations (11.6%). Very few NCBugs are triggered by fresh boot. Hence, we suggest to focus on the following 3 node operations: Crash, shutdown, and reboot.
- The root causes of NCBugs are categorized as follows: 33.3% are distributed concurrency bugs, 36.7% are caused by improper error recovery handling process, and 13.3% are due to incorrect shutdown handler .
- A large amount of NCBugs (18.3%) lead to null pointer exceptions (NPE) and we can develop classic static analysis tools to effectively detect such errors.
- Most NCBugs (85%) need no more than two external requests and events to trigger, and 40% of NCBugs don't need any external requests.
- In contrary to our assumption, it is not complex to fix a NCBug: 72.5% of the patches fix the root cause and another 27.5% patches prevents error propagation.

The above findings shed lights on new techniques to detect and diagnose NCBugs. Based on our findings, we have developed *NPEDetector*, a new static analysis tool to detect those NCBugs leading to null pointer exceptions. NPEDetector detects 65 new bugs from 8 real-world distributed systems, including 7 NCBugs. In summary, this paper makes the following contributions:

- We conducted a comprehensive study of real world NCBugs in 5 representative distributed systems. Our findings help to better understand NCBugs and provide new insights to prevent and diagnose such bugs.
- We developed NCTrigger, a tool to reproduce NCBugs by injecting node operations according to user specification. We have randomly selected 30 bugs and reproduced them with NCTrigger for a deeper analysis of these bugs. NCTrigger can significantly reduce the manual efforts in reproducing a bug, from averagely 2 days to less than half day. It is very helpful in diagnosing the root causes of NCBugs.
- Based on our findings, we developed NPEDetector, a new static analysis tool to detect those NCBugs leading to null pointer exceptions. NPEDetector detects 60 new bugs in 8 distributed systems. We have reported those bugs to their original developers and provided patches for each bug report, 23 bugs have already been fixed by our patches.

TABLE I
SEARCH CRITERIA

Key-Word	Value Criteria
type	"bug", "ask", "sub-task"
status	"Resolved", "Closed"
priority	"Blocker", "Critical", "Major"
resolution	"Fixed"
time	later than 2010-0101
content	contains "kill", "restart", "reboot", "crash", "shutdown", "abort", "down", "dies"

The rest of the paper is organized as follows. Section II presents our empirical study methodology. Section III summarizes the characteristics of NCBugs and answers the above 4 research questions in detail. Section IV discusses challenges and opportunities in detecting NCBugs. Section V introduces our static analysis tool. Section VI reviews related work and Section VII concludes this paper.

II. METHODOLOGY

We select five widely-used real-world open source distributed systems in our study: Scalable file system HDFS [10], scale-out computing framework Hadoop2/Yarn [11], distributed key-value stores HBase [2] and Cassandra [12], and distributed synchronization system Zookeeper [3]. These five systems cover different aspects in distributed computing.

A. Datasets Collection

We collect NCBugs from JIRA [13], the issue repositories of our targeted systems. JIRA allows users to search the historical issues with structured queries. To find out all recent issues that may be caused by node change operations, we perform a key-word based search as in Table I.

The key-word based search generates 6,660 issues, then we manually went through all of them and identified 620 NCBugs. Many issues are not NCBugs. For instance, the issue z-1466 * contains the key-word *"shutdown"*. However, it is just a function name, not related to any node operations. We manually classify those 620 bugs according to their triggering node operations. Table II gives the classification.

TABLE II
DISTRIBUTION OF COLLECTED BUGS.

type	Cassandra	HBase	Hdfs	YARN	Zookeeper	total
shutdown	2	31	14	18	7	72
crash	10	65	22	15	4	116
reboot	109	79	111	97	26	422
fresh boot	9	0	0	0	0	9
Total	130	175	147	133	37	619

As shown in TableII, only 9 bugs are triggered by fresh boot, all of which belong to Cassandra. All systems follow a classic master-slave architecture, except for Cassandra. When a new node joins a master-slave cluster, the master node simply adds it to an available node list. No additional operations are

*the bug id format of JIRA : (system name)-number. We use the initial of the system name for short. To distinguish hbase and hdfs, we use the first two letters

A. Namenode Checkpoint Code

```
1  public void saveDigestAndRenameCheckpointImage(long txid){
2    renameCheckpoint(txid);
3    //reboot here
4    MD5FileUtils.saveMD5File(txid);
5  }
```

B. Checkpoint Workload

```
1  $HADOOP_HOME/bin/hdfs dfs -mkdir /input
2  $HADOOP_HOME/bin/hdfs dfs -put File /input
3  $HADOOP_HOME/bin/hdfs secondarynamenode  -checkpoint force
```

C. Configuration

```
1  bugid = HDFS_4596
2  system_home = $HOME/hdfs-2.0.4-alpha
3  master = hdfs1
4  slaves = hdfs2
5  workload = checkpoint
6  injection = saveDigestAndRenameCheckpointImage 3 reboot
```

D. Framework

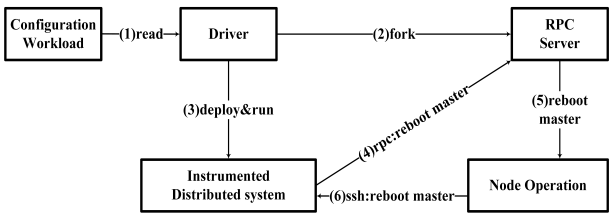

Fig. 1. Reproduce Bug hd-4596 with NCTrigger.

required. Cassandra is a decentralized system. Adding a fresh node will be handled by the complex Gossip protocol [14] implementation, easily causing bugs.

***Finding 1*:** Most NCBugs are triggered by reboot (68.2%), crash (18.7%), or shutdown operations(11.6%). Very few NCBugs are triggered by fresh boot.

B. Analytical Methods

For each node operation, we randomly pick 40 sample bugs for further study. As stated in previous work [15], the sample size 40 is sufficient for our study. Since only few bugs are triggered by fresh boot, we focus on the other three node operations: Shutdown, crash, and reboot.

We manually study all the 120 NCBugs in detail, to answer the above four research questions. Each bug issue is reviewed separately, and cross-validated by two authors. We study each bug issue by carefully reading the issue description, analyzing their related source code, and reviewing their fixing patches. In order to get a deeper understanding of NCBugs, we have randomly selected and reproduced 30 bug examples. We have developed NCTrigger, to help with the process of reproducing. NCTrigger reads user specification from a configuration file (which describes the input and triggering condition for each bug issue), and automatically runs the system to trigger the bug.

1) Example: Figure 1 describes how we reproduce the bug hd-4596 using NCTrigger. The comments in this bug issue clearly point out the bug triggering condition. The workload to trigger this bug issue is given in Figure 1 B. In the source code (Figure 1 A), the variable `txid` represents a `FSImage` file (the meta data of HDFS). The name node will firstly rename the `FSImage` file (line 2), then write its corresponding MD5 file (line 3). If the name node reboots between the two operations, it cannot find the corresponding MD5 file of the renamed `FSImage` file. The reboot process fails.

The configuration file is given in Figure1 C. Users need to specify the location of the system (line 2), the master and slave nodes where the system is deployed (lines 3 and 4), the running workload (line 5), and the source code locations to inject an node operation (Line 6). For each source code location, we need to give the function, the line number, and the event to be injected.

Figure 1 D depicts the workflow of NCTrigger. NCTrigger injects node changing events by instrumenting the targeted distributed systems. The instrumented code will notify an RPC server, which then invokes pre-written scripts to perform corresponding node operations. As shown in the graph, the driver first reads the configuration file. Then it starts the RPC server. Next, it deploys the distributed system and runs the specified workload. We inject node operations (reboot master in this example) by instrumenting the distributed system with a customized *ClassLoader*. The customized *ClassLoader* uses Javaassist [16] to perform instrumentation at the specified source line location. When the system runs the instrumentation code, node operation notifications (e.g., reboot master) are sent to the RPC server, which will then invoke corresponding node operation script. We have written sleep, shutdown, crash and reboot scripts for all target systems. In this example, the reboot script logins to the master via SSH and performs the reboot operation.

2) Experiments: We have reproduced 30 NCBugs using NCTrigger. In our experience, without NCTrigger, it takes averagely 2 days to reproduce a NCBug. With NCTrigger, the average time to reproduce a bug is less than half day. In reproducing the bugs, we also obtain some interesting findings.

Incomplete Comments: 5 bug issues only give the exceptions caused by theses bugs, without information on the source code of their root causes. To reproduce these bugs, we need to thoroughly examine the source code to find the injection point.

Misleading Comments: The bug issue hb-3874 gives a wrong root cause. The suggested injection point will not be executed when running the given workload. We manually identified the correct injection point to reproduce this bug.

Non-Deterministic Reproducing: 3 bugs are reproduced probabilistically because they depend on random values. For example in bug hb-3024, the HMaster node performs recovery when a region server node crashes. The recovery handler reads the meta-data first, which is stored in a randomly-selected node. If the crashed node holds the meta-data, the recovery code will exit (since meta-data is not available) before triggering the bug. We can reproduce these bugs determinately by replacing the random values with deterministic values.

***Finding 2*:** About 30%(9 of 30) NCBugs cannot be reproduced only based on the comments in bug issues.

C. Threats to Validity

The validity of our empirical study may be subject to the threat that we only studied bugs from 5 distributed systems. However, these 5 systems are widely used in practice and they cover various aspects in distributed computing. These systems are the common infrastructures of many applications.

Our study may suffer from errors in manually analyzing each bug. To reduce this threat, each bug issue is reviewed separately by two authors of this paper and we follow the widely-adopted cross-validating method to ensure the correctness of our results. Furthermore, we randomly reproduced 30 bug issues, to further validate our analysis results.

III. Bug Characteristic

We follow the widely-adopted $fault \rightarrow error \rightarrow failure$ model [17] to study each bug issue. Here *fault* is the *root cause*, which can be software bugs, misconfiguration or hardware faults, and *error* is the misbehavior of a system, e.g., null pointer exception. If *error* propagates further, it will result in *failure*, which is observable to the user, e.g., service outage or data loss. For instance, if a null pointer exception error is not caught by the software, it may abort the running node, resulting in the failure of node crash.

For each NCBug, we will study their root causes, the caused errors and their impacts to the system. In addition, we will study how each bug is triggered and fixed.

A. RQ1: Root Cause

1) Distributed Concurrency Bugs: 40 NCBugs are distributed concurrency bugs which involve non-deterministic execution of two threads. Most concurrency bugs are race conditions, i.e., conflicting accesses to shared resources. Files, global variable, and heap objects are common shared resources. For NCBugs, the threads involved include at least one node operation thread (i.e., shutdown thread, crash recover thread or reboot thread). The node operation thread may conflict with a normal user request thread, a daemon thread, or another node operation thread.

There are four different types of distributed concurrency bugs (Figure 2). In contrast to traditional multi-thread software, in distributed systems, the two threads can execute on two different nodes. A thread can access shared resources on another node via inter-node communication such as remote procedure call (RPC).

Order Violation 22 bugs are triggered by wrong orderings of asynchronous operations accessing the same shared resources. Figure 2(a) gives the example y-6168: The resource manager node RM is rebooting and we need to reset the value of the shared memory location. Thread T1 of name node NM will initialize the memory as N via RPC. Meanwhile, thread T2 of application manager AM needs to read the shared memory location via RPC. If T1 initializes the shared memory (W1) before it is being read by T2 (R1), the system behaves correctly. However, if R1 arrives before the shared memory is initialized (W2), T2 reads an uninitialized memory and AM will behave unexpectedly.

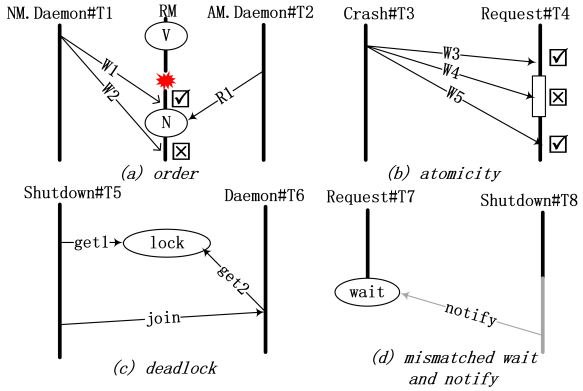

Fig. 2. Four distributed concurrency bug patterns. The vertical lines represent the threads or processes. The arrows represent accesses to shared resources. The ellipses represent the shared resources(e.g. memory, lock). The rectangle represents the code assumed to be atomic but not enforced.

Atomicity Violation 11 bugs manifest because a series of accesses to a shared resource should be regarded as one atomic operation, while accesses to the resource from another thread violates this property. Figure 2(b) shows an atomicity violation example in y-2273. In the resource manager node RM, the request thread T4 performs a deep copy of a shared node map: It firstly stores the set of keys of the shared node map to a local list, then iterates through the local list to get its value from the shard node map. The two operations are assumed to be an atomic operation. However, if a node (N1) crashes, the recovery thread T3 will remove the entry of N1 from the shared node map. If N1 crashes after copying the set of keys to the local list, T4 continues to iterate through the local list and tries to get a node value for N1, a null pointer exception will be thrown.

Deadlock 6 bugs are dead locks. Figure 2(c) shows the example z-2347. When shutting down one node, thread T5 acquires the lock (get1). T5 then waits for T6 to finish (join). Meanwhile, T6 is handling a client request. T6 tries to acquire the same lock (get2) and it can never succeed. As a result, T5 and T6 will wait forever.

Mismatched Wait and Notify There exists one bug (Figure 2(d)). In the shutdown thread(T7), T7 waits for the flush thread T8 to complete. However, T8 meets exception and terminates. As a result, T7 waits for ever and the shutdown process hangs.

2) Improper Recovery Handler: 44 bugs are due to incorrect recovery or reboot handler. When a node reboots or is recovering from crash, the handler needs to reset the node to a proper system state. Bugs may occur if the handler forgets to check and reset the node state, or if it resets the node state with a wrong value, or if it fails to propagate the reset value to the system. We further classified this type of bugs into 6 sub-categories.

Missing Safety Check. 22 bugs manifest because the recovery or reboot handlers access bad values without safety check. For instance, in the bug hb-3023, the HMaster node uses a key-value map to store information of which region

```
1    protected void fetchBlockByteRange(LocatedBlock blockID){
2-     address = read(blockID);
3      while (true) {
4+       address = read(blockID);//miss
5        try{
6           connect(address);
7         }catch(Exception e){//ignore exception and retry};
8      }
9  }
```

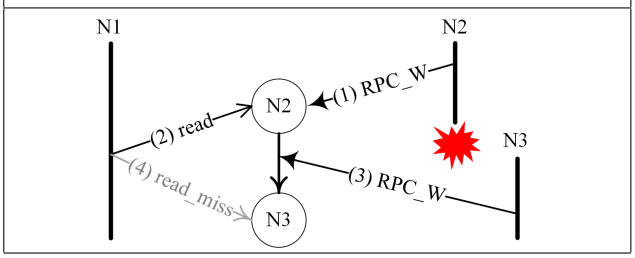

Fig. 3. An example of missing synchronization bug: hd-11708 and related source code. The source code and corresponding patch (on N3) are given.

is stored by which RS node (each region is stored by a corresponding RS node). When HMaster creates a new region, the map is updated by inserting an entry for the newly created region with its RS node value set to null. When a RS node opens the region, the RS node notifies HMaster to update the RS node of the region (in the key-value map) to its address, via RPC. However, if the RS node crashes before the RPC. The recovery handler will read the original null value from the key-value map without checking, resulting in null pointer exception.

In another example (c-11995), the node crashes just after the commit log file is created and its contents is empty. When the node reboots, the reboot handler cannot parse the empty file and reboots fails.

Missing Synchronization. In 9 bugs, the recovery handler has updated the old values correctly. However, the updated new values are not propagated to the system, and the client still uses the old values. For example, in Figure 3, client N1 tries to read a block by executing the function `fetchBlockByteRange`. DataNode N2 notifies N3 that it carries the target block by writing its address to a shared memory (via RPC_W), then N1 reads the address (line 2) and uses it to read blocks from N2 (line 6). If N2 crashes after line 2 and before line 6, the recovery handler will correctly reset the value of the shared memory to the address of a backup node N3 (which also carries the target block). However, client node N1 is not aware of this change. It will keep trying to read blocks from N2 and loop forever.

Reset with Wrong Value. 5 bugs occur because the recovery or reboot threads reset the state with wrong values. As shown in Figure 4, master node N5 writes its address into shared variable `hbaseMaster` (W1). RS node N4 will use this value to connect to the master node (line 3). If the master node crashes, an exception will be thrown and N4 handles the exception by calling the method `getMaster`, to get the new address of master node (line 5). In the normal case, the backup master node N6 resets the value of `hbaseMaster` (W2) to its address, which can be correctly read by N4 via rpc (line 13). However, if the user shuts down N4, the variable `shouldRun`

```
1   public void tryRegionServerReport() {
2     try{
3        this.hbaseMaster.regionServerReport();
4     }catch (IOException ioe) {
5        getMaster();
6     }
7   }
8
9   public void getMaster() {
10    if (!shouldRun) {
11       this.hbaseMaster = null;
12    }else{
13       this.hbaseMaster = rpc();
14    }
15  }
```

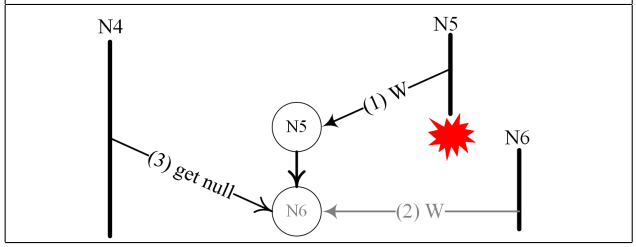

Fig. 4. An example of wrong reset bug hb-10214.

```
1   while (true) {
2     try{
3-       hris = getServer(serverName); //serverName is RS0
4+        // Skip getting user regions if HMaster is stopped.
5+       if (!this.server.isStopped()) {
6+          hris = getServer(serverName);
7+       }
8     } catch (Exception e)( //retry)
9     break;
10  }
```

Fig. 5. hb-10614 source code and patch.

will be set as false. Consequently, `hbaseMaster` is set to null. N4 uses the wrong `hbaseMaster` and raises null pointer exception (line 3).

Excessive Retry. There are 4 bugs in this category. The client node keeps trying to get a value from a crashed node, but always fails. For example in Figure 5, the HMaster node tries to connect to RS0, which holds the meta data (line 3). If RS0 crashes, it will retry until the cluster moves the meta data to a backup node (line 8). However, if the user stops HMaster node, the cluster will not move the meta data and HMaster stucks in retrying. The shutdown process will not terminate.

Missing Recovery. 2 bugs manifest because the reboot handler does not recover system state from the persistence data, e.g. z-1060.

Missing Error Handler. There are 2 bugs where the crash leads to exceptions in the client node, but there are no exception handlers to handle them (e.g. hb-10272).

3) Improper Shutdown Handler: 16 bugs are due to the resources not properly cleaned in shutdown handler. These bugs are classified into 3 sub-categories.

Missing Aftermath Handler. There are 8 bugs in this category. The shutdown handler doesn't close resources (e.g. not closing files) owned by this node (c-3071), or forgets to clean up temporary resources (y-72).

Missing Interrupt. 7 bugs are due to the reason that shutdown handler did not kill user threads (e.g. z-1060). This will make the shutdown process hang. There are two different

types of threads in Java: Daemon threads and user threads. Damon threads will exit automatically when JVM exits, while user threads exits only when it finishes running or is terminated by external signal. If there exists any running user thread, JVM will wait for them to exit.

Called Twice 3 bugs are triggered because the shutdown handler will make a non re-entrant function being called twice, sometimes causing exceptions. For example in y-170, the node manager NM is shutting down and its `stop` method is called directly by the shutdown hook. Meanwhile, another composite service *NodeStatusUpdaterImpl* also notifies the node manager to stop, the `stop` method is called again and an exception is thrown.

4) System-Specific Bugs in Handler.: There are 20 bugs that cannot be classified into the above 3 types. These bugs are system-specific semantic errors in the handler. For example, in m-3463, Yarn uses format *host:port* as id to record each task. However, when the application manager AM crashes, the recovery code uses *host* as id to get tasks. The recover will fail since no tasks can be obtained. 7 bugs happen after upgrading. The updated code cannot parse the old data format, as in c-13559.

***Finding 3*:** The root causes of most NCBugs(83.3%) have common patterns, such as distributed concurrency bugs, or improper recover handler, or fails to clean up resources during shutdown.

B. RQ1: Errors

We categorize the errors of NCBugs into explicit errors and silent errors. Identifying errors can help us understand the root causes of NCbugs, and provide suggestions on how to prevent error propagation.

1) Explicit Errors: 54(45%) NCBugs will print obvious messages, including null pointer exceptions (22 bugs) and other semantic exceptions(32 bugs).

Null Pointer Exception. 22 NCBugs (18.3%) will cause null pointer exceptions(NPE). The ratio is much higher than other types of bugs (5% [5]). In the simple case (8 bugs), the callee method will return a null value when a node is not available (due to shutdown or crash), but the caller does not perform null check on the return value. Another 7 null pointer bugs are due to data races. In the rest 7 null pointer bugs, a shared variable is initialized to null then gets updated by a node. The node may crash before it writes the shared variable, and the recovery thread still uses the initial null value.

Classic static analysis tools can be used to prevent these NCBugs, especially for the simple case. Here we give an example.

```
1   ret = null;//request code
2   while (shouldRun){
3       ret = retry();
4   }
5   return ret;
6   //recovery, reboot, or shutdown code
8   shouldRun = false;
```

The variable `ret` is initialized to null (line 1). It is then updated in the while loop (line 3). Before executing the loop, the reboot or shutdown handler thread may set the variable `shouldRun` to `false`. As a result, the code snippet returns a null value. The caller simply uses the return value without checking. In other cases, an exception is thrown when reading data from a crashing node. Very frequently, the exception handler simply returns a null value, leading to null pointer exception in its caller method.

All five target systems are checked by Findbugs [18], one of the state-of-the-art bug finding tools. However, Findbugs fails to report all the null pointer exception bugs we studied. For inter-procedural bugs, the tool requires developers to add a `Nullable` annotation for the return value of the callee method. Although those annotations are very useful to prevent null pointer bugs, they are not widely adopted in practice. All the five systems in our study do not introduce the *Nullable* annotation.

Nevertheless, we can still detect these null pointer exceptions with traditional static analysis. Here context-sensitive inter-procedural analysis [19]–[22] is required to detect such errors with good precision.

Semantic Exception. The developers will check the correctness of system states by examining values of certain variables. If the values are not expected, exceptions will be thrown with detailed messages printing the diagnosing and locating the root causes. For example in y-1752, due to a concurrency bug, the slave node will send an unexpected event to the master node. The master node checks the event and throws an exception *InvalidStateTransitonException* to indicate that the event is invalid.

2) Silent Errors: 66 bugs do not print explicit error messages, such as system hang or data corruption. There are three types of system hang: (1) File exists in the disk for ever, eventually exhausts resources, (2) System waits for another event or state forever, and (3) Task is too slow and system appears hang.

***Finding 4*:** 45% of NCBugs print explicit error messages, and the other 55% are silent errors.

***Finding 5*:** A large percent of NCBugs lead to null pointer exceptions. These bugs can be effectively detected by traditional static analysis.

C. RQ2: Impact

We classify the impacts of NCBugs into node-, operation-, data- and performance-related failures. Some bugs are innocuous and their errors are tolerated by the system. To avoid double counting, we consider the impact of a bug as operation failures only if the bug does not cause node down, data loss or data corruption.

Node Down. 31 bugs cause node crash or node inoperative. If a bug makes the reboot process fail, it is regarded as causing node crash. We also regard a bug resulting in resource leak as

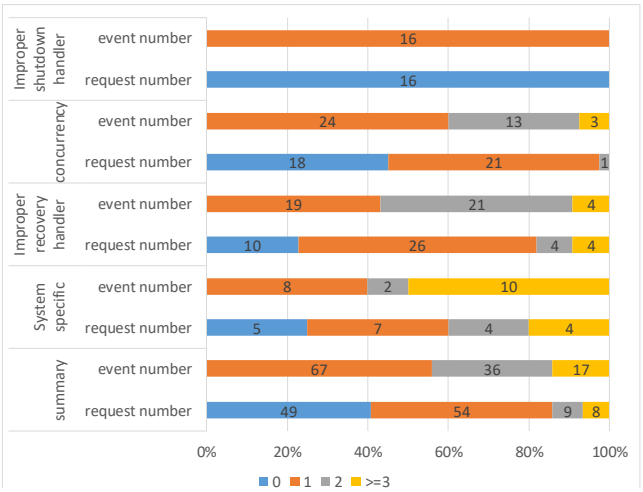

Fig. 6. The distribution of requests and the number of events for each root cause.

TABLE III
THE NUMBER OF REQUEST TYPES IN DIFFERENT SYSTEMS.

HDFS	CASSANDRA	HBASE	YARN	ZK
16	15	13	7	3

causing node inoperative (e.g., y-72), since it will eventually exhaust the disk space.

Data Corruption. 18 bugs lead to data loss or data inconsistencies. For example, in bug hb-3362, two region servers try to update the same data, causing data loss.

Operation Failure. 59 NCBugs cause operation failure. We treat all requests sent to the running system as operations. Hence, failed shutdown and failed client requests are both classified as operation failures.

Performance Degradation. 9 bugs lead to performance degradation, including unnecessary try and wait, or the interval time of trying or waiting takes too long.

Innocuity. 3 bugs have no observable impact to the system. For instance, in y-4355, the null pointer exception crashes the non-service thread. But the error is tolerated by the system.

***Finding 6*:** Allmost all(97.5%) NCBugs have serious impacts to the system, which cause node down, data loss, operation failure, or performance degradation.

D. RQ3: Input Condition

The input condition to trigger NCBugs answers RQ3. The input conditions include the workload (i.e., user requests) and the necessary node operations (i.e., events). Figure 6 summarizes the conditions to trigger a NCBug for each category.

Concurrency Bug. 21 concurrency bugs need one user request to trigger. Among them, 17 bugs need another node operation event (e.g., the example in Figure 2(b)), the rest 4 bugs need two or more node operations. For example, in hb-13546, only when both the master node and the backup node crash, the client request thread will read the wrong value assigned by the recovery thread. 18 concurrency bugs do not need any user request. Among them, 12 bugs need two or more node operations to trigger. For instance, the hb-11458 needs one reboot operation and one shutdown operation to trigger. Before the reboot handler initializes the pointer, the shutdown handler thread begins to execute and reference the uninitialized pointer. 6 bugs need one node operation and another daemon thread trigger. For example in hd-3616, the daemon thread and the shutdown handler thread access the same `HashMap` object which is not thread safe. Hence a *ConcurrentModificationException* is thrown. Only one bug needs two requests and one of the requests will transfer an error into failure.

Improper Recovery Handler. All bugs need at least one node operation (i.e. node crash) to make the system state illegal. The crash can happens on the node which is performing the recovery or shutdown. There are 25 bugs need more than one node operations. There are also 10 bugs do not need any requests to trigger, where the bugs manifest due to conflicting accesses in different threads handling node operations.

Improper Shutdown Handler. All bugs do not need any request, they only need the shutdown operation to trigger.

System-Specific Bug. 5 bugs don't need any request. All of them are due to system upgrades after the node operation. 7 bugs only need one user request. For example, in y-3095, when RM reboots, one variable name becomes different between RM and client node. In another example, m-3463, the bug exists in the event handler, but we need a user request to push the system to a bug-triggering state. 8 bugs need two or more requests to push the system into corner case.

In summary, more than 40% NCBugs don't need any user request, suggesting that all these bugs exist in the operation handlers. 85% NCBugs need no more than two events and requests. Our study also shows that the types of requests can be dispersed, as shown in TableIII. This suggests that in order to automatically trigger node change bugs, we need to inject node operations for different types of requests.

***Finding 7*:** More than 40% NCBugs do not need any request to trigger, and 85% NCBugs need no more than two requests and events.

***Finding 8*:** The types of requests needed to trigger NCbugs are dispersed.

E. RQ4: Fixing Strategies

The fixing strategies reflects the bug complexity. It can also help us investigate bug diagnosis and automatic fix techniques. In studying the fixing strategies of NCBugs, we analyze their patches and related source code. In general, 87 bugs are fixed at the root cause, and 33 bugs are fixed by preventing error propagation.

1) Preventing Error Propagation: Instead of enforcing the thread execution order, 21 concurrency bugs are fixed by preventing the propagation of wrong values. The called twice bugs are fixed by changing the method to be re-entrant, instead of removing one call path. Missing interrupt bugs are not fixed by re-sending the signal, but by adding a variable to record

TABLE IV
COMPLEXITY TO APPLY A PATCH.

Operation	LOC of a patch	# patches	# comments	times (day)
shutdown	45.825	2.98	16.13	83.26
crash	55.98	3.00	20.93	74.49
reboot	50.28	3.50	23.63	87.86
non-NC bugs	61.62	2.81	17.48	82.20

whether the node is down. The running thread then checks this variable accordingly. Two wrong reset bugs are fixed by adding safety check.

2) Fixing the Root Cause: 87 bugs are fixed by addressing their root causes, we describe how the developers fix them for each bug category.

Concurrency Bug. 19 concurrency bugs are fixed this way. In particular, 4 order violation bugs are fixed by enforcing the thread executing order, 3 atomicity violation bugs are fixed by enforcing block synchronization, and 3 deadlock bugs are fixed by removing synchronization. Others are fixed by preventing the interfering thread running. The mismatched wait and notify bug is fixed by adding a Notify operation.

Improper Recovery Handler. All the missing safety check bugs are fixed by adding corresponding safety checks. 3 reset with wrong value bugs are fixed by correcting the wrong values. 4 missing recovery and error handler bugs are fixed by adding the missing handlers. Missing synchronization bugs are fixed by adding the retry code. For example, in figure3, the patch is to move the read code into while block. Excessive retry bugs are fixed by adding code to check the client state.

System-Specific Bugs. All the System-specific bugs are fixed by addressing the wrong code. For upgrade bugs, the fixing is improving the new code to match the old data format.

3) Fixing Complexity: Is it more complex to fix NCBugs than other types of bugs? In order to measure their fixing complexity, we random chose 120 non-NCBugs and compare their patches with those of NCBugs. We compare the numbers of patched code lines (including deleted and added lines), the number of patches, and the number of days to apply a patch. Before the bug issue is closed, the developers will discuss the root causes and fixing strategies in the issue repositories. The number of comments suggests the complexity in diagnosing and fixing a bug. Hence, we also compare the number of comments for each bug. We have developed a python script to parse the issue web page to get the above statistics.

As shown in Table IV, surprisingly, NCBugs are no harder to fix than other types of bugs. Shutdown bugs are generally easier to fix, since those bugs commonly happen in the shutdown handler only. We assume that the reason is that tolerating node change is implemented as normal software components of distributes system. Hence, the complexity of NCBugs does not introduce extra complexity in their fixes.

***Finding 9*:** 72.5% of NCBugs are fixed by fixing the root cause, the other NCBugs are fixed by preventing error propagation.

***Finding 10*:** It is no more complex to fix a NCBug than other types of bugs. Shutdown bugs are easier to fix.

IV. LESSONS LEARNED

In this section, we discuss challenges and opportunities to prevent and detect NCBugs.

A. Opportunities

Shutdown Bug Detection. Finding 3 shows that graceful shutdown can also cause bugs and these bugs are generally easy to detect and fix. According to Finding 6, shutdown bugs also have serious impacts to the system. However, current studies mostly focus on detecting bugs caused by crash and reboot [6], [9], which is more challenging. New analysis and testing tools can be developed to effectively address shutdown bugs.

Concurrency Bug Detection. Concurrency bugs caused by node operation have similar patterns to traditional concurrency bugs. We can extend existing detection and testing techniques [23]–[26] for distributed concurrency bugs to detect this type of bugs.

Static Analysis Tools. Our study shows that 27.5% bugs are fixed to prevent error propagation (Finding 9). This finding suggests that we can detect the bugs based on their error propagation rule, similar to previous work [15], [27]. For example, those null point exception bugs triggered by node operations (Finding 5) share a simple rule. Although existing static analysis tools cannot detect them, new rule-based tools without complex analysis can be developed.

Model Checking Tools. Model checking tools detect bugs by enumerating all events order, and it faces the state space explosion when a large number events are involved. Finding 7 show that 85% bugs don't need more than 2 events, suggesting new optimization opportunities to prune unnecessary events.

B. Challenges

Inferring Request. Finding 7 shows that the types of requests trigging node operation bugs are dispersed. The node operation handler and different request handler threads interact with each other through shared resources. However, it is difficult to infer which user requests are needed to trigger those bugs. For instance, some bugs are only triggered when the system is being upgraded.

Fault Injection. Existing fault injection techniques inject faults (e.g., crash events) at IO or communication points. However, our study shows that certain bugs are triggered by node operations at other program points. Hence, we should develop the new technology that can explore more injection points. The space we need to explore to inject faults becomes much larger.

V. NPEDETECTOR

Static analysis tools [18], [28], [29] are now commonly used in large software projects to detect the potential null pointer exceptions (NPE) at compile time. All the five target systems in our study have Findbugs [18] as a built-in tool. These static analysis tools are effective. For example, Nullaway [28] has helped Uber reduce the number of NPEs in production by an order of magnitude [30]. Due to restrictions on analysis

TABLE V
RESULTS OF APPLYING NPEDETECTOR TO 8 DISTRIBUTED SYSTEMS.

System	YARN	HDFS	HBase	Cassandra	Zookeeper	CloudStack	Storm	Helix	total
submitted bugs	4	9	11	3	12	12	9	5	65
confirmed bugs	0	7	8	0	1	0	5	2	23
false positive	0	0	1	0	0	1	0	3	5
uncertainty	0	2	0	3	8	0	1	0	14
fixed bugs	4	0	2	0	3	11	3	0	23

efficiency, these static analysis tools commonly do not perform complex inter-procedural analysis. Hence, they rely on developers to add a `@Nullable` annotation for each method which may return null. They can then efficiently check whether the return value of a `@Nullable` method is dereferenced without checking or not.

It is tedious for developers to introduce `@Nullable` annotations, especially for large projects with millions of lines of code. This largely restricts the ability and effectiveness of existing static analysis tools. Unfortunately, there is no effective tool to help developers to introduce the `@Nullable` annotations automatically.

Our finding 5 shows that those null pointer exceptions caused by node operations share a simple pattern: 1) the callee method directly returns a null value in its exception handler (due to crash or other node operations), and 2) caller directly uses the null value without checking. Base on this observation, we develop NPEDetector, a rule-based static analysis tool to automatically locate nullable methods.

A. Implementation

NPEDetector is implemented on top of the WALA static analysis framework [31] (v1.4.3). The tool firstly finds all methods that can return a null value directly, i.e., has a `return null` instruction. Then it gives each method that may return a null value a score to measure its nullable probability. Specifically, we use the following formula to compute the probability.

$$Score = CheckedCallersN - UnCheckedCallersN + \#Exception * Weight \quad (1)$$

The probability score depends on how much callers have checked the return value of this method. The more the number of callers checking (not checking) its return value, the higher (lower) the score. In addition, since node operations often cause exceptions, we put more emphasis on those methods which return a null value in their exception handlers. The score is adjusted by $\#Exception * weight$, where $\#Exception$ is the number of exception handlers, and $Weight$ is pre-defined a constant value. By default, it is set to 10.

Next, we rank all methods according to their probability scores. We implement a simple checker which regards an unchecked reference in the caller of a nullable method as a bug, and report those bugs with highest scores.

Discussion: There are different use cases of NPEDetector. Instead of directly reporting null pointer bugs, we can report the top ranked methods to the developers for manual inspection. Alternatively, we can automatically introduce `@Nullable` comments and run existing static tools for bug reporting. Our current implementation targets Java, but the intuition is applicable for programs in other languages.

```
1  //caller
2  public list getChildData(String znode) {
3    List<String> nodes = listChildren();
4    for (String node :nodes){//other code here}
5  }
6  //callee
7  public List listChildren() {
8    try{
9        return zkw.getChildren();
10    } catch(NoNodeException ke){
11       return null;
12    }
13 }
```

Fig. 7. The new bug hb-20419 that is caused by crash.

```
1  //shutdown
2  public list setAMContainerSpec() {
3    clearAmContainerSpec();
4  }
5  //caller
6  public List createAMContainerLaunchContext() {
7    getAMContainerSpec().setTokens();
8  }
9  //callee
10 public List getAMContainerSpec() {
11   if (!hasAmContainerSpec()) {
12      return null;
13   }
14 }
```

Fig. 8. The new bug y-7786 that is caused by shutdown.

B. Checking Real-World System

We have applied NPEDetector to analyze the latest versions of 8 distributed systems. In addition to the five target systems in our study, we have also chosen 3 new distributed systems: Cloud computing platform CloudStack [32], distributed realtime computation system Storm [33], and generic cluster management framework Helix [34]. This can help to evaluate the generality of NPEDetector. In our experiments, we report the top 100 bugs with highest scores and manual inspect each bug report.

There are 35 false positives where the callers already perform safety check before calling nullable method. We believe that the rest 65 bug reports are real bugs. We have submitted them to the original developers and provided patch for each submitted bug (Table V). Overall, NPEDetector has found 60 new null pointer exception bugs. Among the 60 bugs, 23 bugs have already been fixed using our provided patches, 23 bugs have already been confirmed but not fixed yet, 14 bugs are still waiting to be reviewed. For the 3 newly added distributed systems, NPEDetector reports 26 bugs in total, including 14 bugs which are already fixed. For CloudStack, 11 of 12 our

submitted bugs have been fixed. Our provided patches have also been committed to the milestone version by the origin developer.

Ability of Detecting NCBugs. NPEDetector reports 7 null pointer exception bugs caused by node operations, all of which have been fixed. Figure 7 shows one new bug caused by node crash. The caller method `getChildData` invokes the callee method `listChildren` to get the list of nodes (line 3). The `listChildren` method connects to `zkw` (line 9). If the node crashes, callee will return a null value directly in its exception handler (line 11). The caller method `getChildData` directly uses the null value without checking (line 4), leading to a null pointer exception error.

Figure 8 shows another example caused by shutdown. The application manager node AM is shutting down and it will clean all its resource (line 3). As a result, the condition at line 11 is evaluated to be true, the callee method `getAMContainerSpec` directly returns a null value. In line 7, the null value is directly used without checking. This bug is a data race and we have triggered it with NCTrigger.

Performance. The analysis is very efficient and most of the time is spent on building the call graph by WALA. It costs less than 24 seconds for our analysis to process YARN (with millions lines of code) on an Intel(R) Xeon(R) E7-4809 processor with 32 GB of memory.

False Positives. 5 of our submitted bugs are confirmed as false positives. 1 bug is due to the fact that the caller is depreciated, and that method has been deleted in our accepted patch (cl-10356). In other cases, the return value are guaranteed to be not null with complex logic instead of being checked directly (e.g. he-702).

Feedback from Developer. Our patches and tool have received many positive feedbacks, with comments including *"Good catch!", "Thanks for the nice contribution, I really appreciate your enthusiasm to develop such a handy tool"*.

VI. Related Work

In this section we will introduce some distributed bug studies and detection technologies which are related to our work.

Distributed System Bugs Studies. There are many empirical studies on distributed system bugs [5], [15], [35]–[40]. Some of them only focus on one system. Due to space constraints, here we only discuss the bug studies that based on the multi open source distributed systems. CREB [41] gives a nice empirical study about bugs caused by node crashes. CBSDB [38] performs a monolithic distributed system bug study and gives many interesting findings, such as 13% bugs are caused by hardware faults(including node crash). We concentrate on NCBugs and deeply study them along with source code and patches. TaxDC [5] performs a study on distributed concurrency bugs and finds that 63% distributed concurrency bugs need node crash, reboot, and other faults as input. Our study focuses on NCBugs, concurrency bugs are only part of them. Yuan [15] found that 35% the catastrophic failures are caused by inappropriate error handler and also developed a tool to detect such bugs. We also find a simple rule of NPEs and develop a tool to prevent them based on the Finding 5.

Distributed System Bugs Detection. Previous works [6]–[8], [42], [43] focus on injecting faults(include node crash, disk failure) into running system. FATE [6] systematically pushes system into many possible failure scenarios by abstracting IO state. Setsudo [8] injects fault while system is in recovery state. Ju [7] first constructs the system execution graph based on the internetwork communication and injects fault at each graph node. Our study shows that node crash which happens at more than IO point can also cause bugs. Model checking [9], [44]–[47] systematically enumerates the possible event order of a distributed system and performs some fault injections at a special system state, this makes the system run into corner-case situations and exposes deep hidden bugs. Model checking may enumerate many unnecessary events and hence take the edge off its performance, our study shows that 85% NCBugs involve no more than 2 events. We still have chance to reduce the number of events that are needed to be enumerated. DCatch [48] is the first work that detects the data race bugs in distributed system, but has limit on detecting concurrency typestate bugs [49]. CloudRaid [50] can detect distributed concurrency bugs by log mining. FCatch [51] detects fault timing bug by brilliantly transforming the bugs into two common models. DScope [27] finds the data corruption bugs which can led hang by static analysis. PCatch [52] first identifies the local slowdown region, then analyses the region propagation chain to make sure whether it will affect all system performance. Our study also covers data race, fault timing, data corruption and performance bugs, we believe that our findings can be helpful to improve these detection tools.

VII. Conclusion and Future Work

Distributed systems have become the basic platform to support many other applications, but they still suffer from the NCBugs. In this paper, we choose 120 real world NCBugs from 5 widely used distributed systems and perform comprehensive study on them. We examine their root cause, impact, input condition, and fixing strategies. Our findings can promote future NCBug detection. Also we develop a tool to detect bugs in 8 distributed systems. In total 60 bugs are found and 23 of them are fixed by our patches. In the future, we will develop a more powerful toolkit for detecting NCBugs in distributed system.

The source code of NPEDetector, new found bugs and discussions with the original developers can be found at:

https://github.com/lujiefsi/NPEDetector

Acknowledgement

This work is supported by the National Key research and development program of China (2016YFB1000402), the Innovation Research Group of National Natural Science Foundation of China (61521092 and 61672492), and the National Natural Science Foundation of China (U1736208).

REFERENCES

[1] iwgcr. (7/mar/2012) Downtime costs per hour. [Online]. Available: http://iwgcr.org/?p=404

[2] L. George, *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9. Boston, MA, USA, 2010.

[4] J. Tsidulko, ""microsoft azure storage cluster loses power"," Aug. 2017. [Online]. Available: https://www.crn.com/slide-shows/cloud/300089786/the-10-biggest-cloud-outages-of-2017-so-far.htm/pgno/0/6

[5] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 517–530, 2016.

[6] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "Fate and destini: A framework for cloud recovery testing," in *Proceedings of NSDI 11: 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011, p. 239.

[7] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva, "On fault resilience of openstack," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 2.

[8] P. Joshi, M. Ganai, G. Balakrishnan, A. Gupta, and N. Papakonstantinou, "Setsudō: perturbation-based testing framework for scalable distributed systems," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*. ACM, 2013, p. 7.

[9] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems." in *OSDI*, 2014, pp. 399–414.

[10] D. Borthakur *et al.*, "Hdfs architecture guide," *Hadoop Apache Project*, vol. 53, 2008.

[11] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[12] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[13] T. A. S. Foundation, "Asf jira - apache issues," 2018. [Online]. Available: https://issues.apache.org/jira/

[14] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized gossip algorithms," *IEEE transactions on information theory*, vol. 52, no. 6, pp. 2508–2530, 2006.

[15] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems." in *OSDI*, 2014, pp. 249–265.

[16] S. Chiba, "Load-time structural reflection in java," in *European Conference on Object-Oriented Programming*. Springer, 2000, pp. 313–336.

[17] J.-C. Laprie, "Dependable computing: Concepts, limits, challenges," in *Special Issue of the 25th International Symposium On Fault-Tolerant Computing*, 1995, pp. 42–54.

[18] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.

[19] L. Li, C. Cifuentes, and N. Keynes, "Boosting the performance of flow-sensitive points-to analysis using value flow," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 343–353.

[20] L. Lian, C. Cifuentes, and N. Keynes, "Precise and scalable context-sensitive pointer analysis via value flow graph," in *Proceedings of the 2013 International Symposium on Memory Management*, ser. ISMM '13. New York, NY, USA: ACM, 2013, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/2464157.2466483

[21] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 460–473.

[22] T. Tan, Y. Li, and J. Xue, "Efficient and precise points-to analysis: modeling the heap by merging equivalent automata," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 278–291, 2017.

[23] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 121–133.

[24] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race detection for event-driven mobile applications," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 326–336, 2014.

[25] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. data race bugs: telling the difference with portend," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 185–198, 2012.

[26] Q. Zhou, L. Li, L. Wang, J. Xue, and X. Feng, "May-happen-in-parallel analysis with static vector clocks," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 228–240.

[27] T. Dai, J. He, X. Gu, S. Lu, and P. Wang, "Dscope: Detecting real-world data corruption hang bugs in cloud server systems," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2018, pp. 313–325.

[28] Nullaway: Fast annotation-based null checking for java. [Online]. Available: https://github.com/uber/NullAway

[29] (2018) Infer : Eradicate. [Online]. Available: http://fbinfer.com/docs/eradicate.html

[30] Engineering nullaway, uber's open source tool for detecting nullpointerexceptions on android. [Online]. Available: https://eng.uber.com/nullaway/

[31] T. W. Libraries, "Wala." [Online]. Available: https://github.com/wala/WALA

[32] (2018) cloudstack. [Online]. Available: https://cloudstack.apache.org/

[33] (2018) storm. [Online]. Available: http://storm.apache.org/

[34] (2018) helix. [Online]. Available: https://engineering.linkedin.com/apache-helix/apache-helix-framework-distributed-system-development

[35] S. Li, H. Zhou, H. Lin, T. Xiao, H. Lin, W. Lin, and T. Xie, "A characteristic study on failures of production distributed data-parallel programs," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 963–972.

[36] P. Wang, D. J. Dean, and X. Gu, "Understanding real world data corruptions in cloud systems," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015, pp. 116–125.

[37] M. R. Mesbahi, A. M. Rahmani, and M. Hosseinzadeh, "Cloud dependability analysis: Characterizing google cluster infrastructure reliability," in *Web Research (ICWR), 2017 3th International Conference on*. IEEE, 2017, pp. 56–61.

[38] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin *et al.*, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[39] A. Rabkin and R. Katz, "How big hadoop clusters break in the real world."

[40] T. Xiao, J. Z. H. Z. Z. Guo, S. McDirmid, and W. L. W. C. L. Zhou, "Nondeterminism in mapreduce considered harmful?" 2014.

[41] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu, "An empirical study on crash recovery bugs in large-scale distributed systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 539–550.

[42] F. Dinu and T. Ng, "Understanding the effects and implications of compute node related failures in hadoop," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012, pp. 187–198.

[43] P. Joshi, H. S. Gunawi, and K. Sen, "Prefail: A programmable tool for multiple-failure injection," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 171–188.

[44] H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "Modist: Transparent model checking of unmodified distributed systems," in *6th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2009.

[45] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang, "Practical software model checking via dynamic interface reduction," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 265–278.

[46] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code." NSDI, 2007.

[47] J. Simsa, R. E. Bryant, and G. Gibson, "dbug: systematic evaluation of distributed systems." USENIX, 2010.

[48] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "Dcatch: Automatically detecting distributed concurrency bugs in cloud systems," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 677–691.

[49] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, "2ndstrike: toward manifesting hidden concurrency typestate bugs," *ACM Sigplan Notices*, vol. 47, no. 4, pp. 239–250, 2012.

[50] J. Lu, F. Li, L. Li, and X. Feng, "Cloudraid: hunting concurrency bugs in the cloud via log-mining," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 3–14.

[51] H. Liu, X. Wang, G. Li, S. Lu, F. Ye, and C. Tian, "Fcatch: Automatically detecting time-of-fault bugs in cloud systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 419–431.

[52] J. Li, Y. Chen, H. Liu, S. Lu, Y. Zhang, H. S. Gunawi, X. Gu, X. Lu, and D. Li, "Pcatch: automatically detecting performance cascading bugs in cloud systems," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 7.