TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems

Tanakorn Leesatapornwongsa
University of Chicago
tanakorn@cs.uchicago.edu

Jeffrey F. Lukman
Surya University
lukman@cs.uchicago.edu

Shan Lu
University of Chicago
shanlu@uchicago.edu

Haryadi S. Gunawi
University of Chicago
haryadi@cs.uchicago.edu

Abstract

We present TaxDC, the largest and most comprehensive taxonomy of non-deterministic concurrency bugs in distributed systems. We study 104 distributed concurrency (DC) bugs from four widely-deployed cloud-scale datacenter distributed systems, Cassandra, Hadoop MapReduce, HBase and ZooKeeper. We study DC-bug characteristics along several axes of analysis such as the triggering timing condition and input preconditions, error and failure symptoms, and fix strategies, collectively stored as 2,083 classification labels in TaxDC database. We discuss how our study can open up many new research directions in combating DC bugs.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Diagnostics

Keywords Concurrency bugs, distributed systems, software testing

1. Introduction

> *"Do we have to rethink this entire [HBase] root and meta 'huh hah'? There isn't a week going by without some new bugs about* ***races*** *between splitting and assignment [distributed protocols]." — h4397*

Software systems are getting more complex and new intricate bugs continue to appear, causing billions of dollars in economic loss. One notorious type of software bugs is concurrency bugs. These timing-related bugs manifest non-deterministically, and hence are extremely difficult to detect, diagnose, and fix. A huge body of work exists in this space that focuses on "local" concurrency (LC) bugs in single-machine multi-threaded software, caused by incorrect interleaving of memory accesses.

Today, beyond single-machine software, distributed software infrastructures have become a dominant backbone for cloud computing and modern applications. Large-scale distributed systems such as scalable computing frameworks [4, 14], storage systems [3, 6, 11, 15], synchronization [7, 10] and cluster management services [5, 29] have emerged as the datacenter operating system. Increasing numbers of developers write large-scale distributed systems and billions of end users rely on the reliability of these systems.

Unfortunately, the reliability of datacenter distributed systems is severely threatened by non-deterministic concurrency bugs as well, which we refer as *distributed concurrency (DC) bugs*. Distributed systems execute many complicated distributed protocols on hundreds/thousands of machines with no common clocks, and must face a variety of random hardware failures [17, 25]. This combination makes distributed systems prone to DC bugs caused by non-deterministic timing of distributed events such as message arrivals, node crashes, reboots, and timeouts. These DC bugs cannot be directly tackled by LC bug techniques, and they cause fatal implications such as operation failures, downtimes, data loss and inconsistencies.

Fighting DC bugs is challenging, particularly given the preliminary understanding of real-world DC bugs. To make progress, we believe a comprehensive bug study is needed. Past studies have closely examined bugs in various software systems [12, 44, 50], which have motivated and guided many aspects of reliability research. There are few bug studies on large-scale distributed systems [25, 39], but they did not specifically dissect DC bugs. One recent paper ana-

ASPLOS '16, April 02 - 06, 2016, Atlanta, GA, USA.

ACM 978-1-4503-4091-5/16/04. . . $15.00.
http://dx.doi.org/10.1145/2872362.2872374

lyzed non-determinism in MapReduce programs but only discussed five bugs [64]. Thorough studies have also been conducted for LC bugs [21, 45] with many follow-up work to date. To the best of our knowledge, *there is no comprehensive study on real-world distributed concurrency bugs*.

A comprehensive study on real-world DC bugs relies on three key ingredients: detailed bug descriptions by users and developers, open access to source code and patches, and pervasive documentation of the studied systems. Although DC bugs theoretically have existed for decades, all the key ingredients were never aligned, causing the lack of comprehensive DC bug studies. There were only a few popular open-source distributed systems [1, 37, 60], and they do not document bugs in detail. There were a few bug detection and diagnosis papers that discussed DC bugs in real-world proprietary systems [26, 42, 43, 55, 65], but the number of reported bugs is too few for a comprehensive study and they do not release source code and patches. Fortunately, in the advent of open-source cloud computing, many of today's popular datacenter distributed systems provide all the key ingredients.

1.1 TaxDC

This paper presents our in-depth analysis of 104 DC bugs. We believe our study is the largest and most comprehensive to date for DC bugs. The bugs came from four popular datacenter distributed systems: Cassandra [3], HBase [6], Hadoop MapReduce [4], and ZooKeeper [7]. We introduce TaxDC, a comprehensive taxonomy of real-world DC bugs across several axes of analysis such as the triggering timing condition and input preconditions, error and failure symptoms, and fix strategies, as shown in detail in Table 1. The results of our study are stored in the form of 2,083 classification labels in TaxDC database [2].

As our main contribution, TaxDC will be the first large-scale DC-bug benchmark. In the last five years, bug benchmarks for LC bugs have been released [32, 66], but no large-scale benchmarks exist for DC bugs. Researchers who want to evaluate the effectiveness of existing or new tools in combating DC bugs do not have a benchmark reference. TaxDC provides researchers with more than 100 thoroughly taxonomized DC bugs to choose from. Practitioners can also use TaxDC to check whether their systems have similar bugs. The DC bugs we studied are considerably general, representing bugs in popular types of distributed systems.

As a side contribution, TaxDC can help open up new research directions. In the past, the lack of understanding of real-world DC bugs has hindered researchers to innovate new ways to combat DC bugs. The state of the art focuses on three lines of research: monitoring and post-mortem debugging [22, 42, 43, 55], testing and model checking [26, 37, 38, 59, 65], and verifiable language frameworks [16, 63]. We hope our study will not only improve these lines of research, but also inspire new research in bug detection tool design, runtime prevention, and bug fixing, as elaborated more in Section 8.

1.2 Summary of Findings

While we provide detailed findings throughout the paper, we first summarize findings on the intricacies of DC bugs:

• Throughout the development of our target systems, new DC bugs continue to surface. Although these systems are popular, there is a lack of effective testing, verification, and analysis tools to detect DC bugs prior to deployment.

• Real-world DC bugs are hard to find because many of them linger in complex concurrent executions of *multiple* protocols. Complete systems contain many background and operational protocols beyond user-facing foreground protocols. Their concurrent interactions can be deadly.

• 63% of DC bugs surface in the presence of hardware faults such as machine crashes (and reboots), network delay and partition (timeouts), and disk errors. As faults happen, recovery protocols create more non-deterministic events concurrent with ongoing operations.

• 47% of DC bugs lead to silent failures and hence are hard to debug in production and reproduce offline.

Nevertheless, through a careful and detailed study of each bug, our results also bring fresh and positive insights:

• More than 60% of DC bugs are triggered by a *single* untimely message delivery that commits order violation or atomicity violation, with regard to other messages or computation. This finding motivates DC bug detection to focus on timing-specification inference and violation detection; it provides simple program-invariant and failure-predictor templates for DC bug detection, failure diagnosis, and runtime prevention.

• 53% of DC bugs lead to *explicit* local or global errors. This finding motivates inferring timing specifications based on local correctness specifications, in the form of error checking already provided by developers.

• Most DC bugs are fixed through a small set of strategies. 30% are fixed by prohibiting the triggering timing and another 40% by simply delaying or ignoring the untimely message, or accepting it without introducing new handling logic. This finding implies unique research opportunities for automated in-production fixing for DC bugs.

• Many other observations are made that enable us to analyze the gap between state-of-the-art tools and real-world DC bugs as well as between research in LC and DC bugs.

In the following sections, we first present our methodology (§2) and then our DC taxonomy (§3-§7). Note that these sections only present the taxonomy, illustrations, examples, and statistics. Later, Section §8 discusses the lessons learned wherein we will connect our findings with the implications to existing tools and opportunities for new research in combating DC bugs. We do not have an exclusive section for related work as we discuss them throughout the paper.

2. Methodology

2.1 Basic Definitions

A *distributed concurrency (DC) bug* is a concurrency bug in distributed systems caused by distributed events that can occur in non-deterministic order. An *event* can be a message arrival/sending, local computation, fault, and reboot. A *local concurrency (LC) bug* is a concurrency bug that happens locally within a node due to thread interleaving. In our model, a *distributed system* is a collection of shared-nothing nodes. Each node can run multiple protocols in multiple threads.

2.2 Target Systems and Dataset

Our study examined bugs from four widely-deployed open-source datacenter distributed systems that represent a diverse set of system architectures: Hadoop MapReduce (including Yarn) [4] representing distributed computing frameworks, HBase [6] and Cassandra [3] representing distributed key-value stores (also known as NoSQL systems), and ZooKeeper [7] representing synchronization services. They are all fully complete systems containing many complex concurrent protocols. Throughout the paper, we will present short examples of DC bugs in these systems. Some detailed examples are illustrated in Figure 2, 4 and 5.

The development projects of our target systems are all hosted under Apache Software Foundation wherein organized issue repositories (named "JIRA") are maintained. To date, across the four systems, there are over 30,000 issues submitted. One major challenge is that issues pertaining to DC bugs do not always contain plain terms such as "concurrency", "race", "atomicity", etc. Scanning all the issues is a daunting task. Thus, we started our study from an open source cloud bug study (CBS) database [2], which already labels issues related to concurrency bugs. However, beyond simple labeling, the CBS work did not differentiate DC from LC bugs and did not dissect DC bugs further.

From CBS, we first filtered out LC bugs, then exclude DC bugs that do not contain clear description, and finally randomly picked 104 samples from the remaining detailed DC bugs, specifically 19 Cassandra, 30 HBase, 36 Hadoop MapReduce, and 19 ZooKeeper DC bugs, reported in January 2011-2014 (the time range of CBS work). We have seen much fewer clearly explained DC bugs in CBS from Cassandra and ZooKeeper than those from HBase and Hadoop MapReduce, which may be related to the fact that they are different types of distributed systems. For example, ZooKeeper, as a synchronization service, is quite robust as it is built on the assumption of event asynchrony since day one. Cassandra was built on eventual consistency, and thus did not have many complex transactions, until recently when Cassandra adopts Paxos. We still see new DC bugs throughout 2014-2015 (some pointed to us by the developers); they can be included into TaxDC in the future.

Triggering (§3)
What is the triggering timing condition?
Message arrives unexpectedly late/early
Message arrives unexpectedly in the middle
Fault (component failures) at an unexpected state
Reboot at an unexpected state
What are the triggering inputs preconditions?
Fault, reboot, timeout, background protocols, and others
What is the triggering scope?
How many nodes/messages/protocols are involved?
Errors & Failures (§4)
What is the error symptom?
Local memory exceptions
Local semantic error messages & exceptions
Local hang
Local silent errors (inconsistent local states)
Global missing messages
Global unexpected messages
Global silent errors (inconsistent global states)
What is the failure symptom?
Node downtimes, data loss/corruption, operation failures, slowdowns
Fixing (§5)
What is the fix strategy?
Fix Timing: add global synchronization
Fix Timing: add local synchronization
Fix Handling: retry message handling at a later time
Fix Handling: ignore a message
Fix Handling: accepting a message without new computation logics
Fix Handling: others

Table 1. Taxonomy of DC Bugs.

2.3 Taxonomy

We study the characteristics of DC bugs along three key stages: triggering, errors & failures, and fixing (Table 1). *Triggering* is the process where software execution states deviate from correct to incorrect under specific conditions. At the end of this process, the manifestation of DC bugs changes from non-deterministic to deterministic. *Errors and failures* are internal and external software misbehaviors. *Fixing* shows how developers correct the bug. We will discuss in detail these categories in their respective sections.

2.4 Threats to Validity

For every bug, we first ensure that the developers marked it as a real bug (not a false positive). We also check that the bug description is clear. We then *re-enumerate* the full sequence of operations (the "*steps*") to a clearer and more concise description such as the ones in Figure 2. Finally, to improve the quality of the taxonimization process, each bug classification is reviewed by *all* authors in this paper. Our study cannot and does not cover DC bugs not fixed by the developers. Even for fixed bugs, we do not cover those that are not described clearly in the bug repositories, a sacrifice we had to make to maintain the accuracy of our results.

Readers should be cautioned not to generalize the statistics we report as each distributed system has unique purpose, design and implementation. For example, we observe 2:1 overall ratio between order and atomicity violations §3.1,

however the individual ratios are different across the four systems (*e.g.* 1:2 in ZooKeeper and 6:1 in MapReduce). Like all empirical studies, our findings have to be interpreted with our methodology in mind.

2.5 TaxDC Database

We name the product of our study TaxDC database. TaxDC contains in total 2,083 classification labels and 4,528 lines of clear and concise re-description of the bugs (our version, that we manually wrote) including the re-enumeration of the steps, triggering conditions, errors and fixes. We will release TaxDC to the public [1]. We believe TaxDC will be a rich "bug benchmark" for researchers who want to tackle distributed concurrency problems. They will have sample bugs to begin with, advance their work, and do not have to repeat our multi-people-year effort.

2.6 Detailed Terminologies

Below are the detailed terminologies we use in this paper. We use the term "state" to interchangeably imply *local state* (both in-memory and on-disk per-node state) or *global state* (a collection of local states and outstanding messages). A *protocol* (*e.g.*, read, write, load balancing) creates a chain of events that modify system state. User-facing protocols are referred as *foreground* protocols while those generated by daemons or operators are referred as *background* protocols.

We consider four types of *events*: message, local computation, fault and reboot. The term *fault* represents component failures such as crashes, timeouts, and disk errors. A *timeout* (system-specific) implies a network disconnection or busy peer node. A *crash* usually implies the node experiences a power failure. A *reboot* means the node comes back up.

Throughout the paper, we present bug examples by abstracting system-specific names. As shown in Figure 1, we use capital letters for nodes (*e.g.*, A, B), two small letters for a message between two nodes (*ab* is from A to B). Occasionally, we attach system-specific information in the subscript (*e.g.*, $A_{\text{AppMaster}}$ sends ab_{taskKill} message to $B_{\text{NodeManager}}$). We use "/" to imply concurrency (ac/bc implies the two messages can arrive at C in different orders, *ac* or *bc* first). A dash, "–", means causal relation of two events (*ab*-*bc* means *ab* causally precedes *bc*). Finally, we use "N*" to represent crash, "N!" reboot, and "N+" local computation at N.

We cite bug examples with clickable hyperlinks (*e.g.*, m3274). To keep most examples uniform, we use MapReduce examples whenever possible. We use the following abbreviations for system names: "c/CA" for Cassandra, "h/HB" for HBase, "m/MR" for Hadoop MapReduce, and "z/ZK" for ZooKeeper; and for system-specific components: "AM" for application master, "RM" for resource manager, "NM" for node manager, "RS" for region server, and "ZAB" for ZooKeeper atomic broadcast.

[1] Please check our group website at `http://ucare.cs.uchicago.edu`

3. Trigger

"That is one monster of a race!" — m3274

DC bugs often have a long triggering process, with many local and global events involved. To better reason about this complicated process, we study them from two perspectives:

1. *Timing conditions* (§3.1): For every DC bug, we identify the smallest set of concurrent events E, so that a specific ordering of E can guarantee the bug manifestation. This is similar to the interleaving condition for LC bugs.
2. *Input preconditions* (§3.2): In order for those events in E to happen, regardless of the ordering, certain inputs or fault conditions (*e.g.*, node crashes) must occur. This is similar to the input condition for LC bugs.

Understanding the triggering can help the design of testing tools that can proactively trigger DC bugs, bug detection tools that can predict which bugs can be triggered through program analysis, and failure prevention tools that can sabotage the triggering conditions at run time.

3.1 Timing Conditions (TC)

Most DC bugs are triggered either by untimely delivery of messages, referred to as *message timing bugs*, or by untimely faults or reboots, referred to as *fault timing bugs*. Rarely DC bugs are triggered by both untimely messages and untimely faults, referred to as *message-fault bugs*. Table 2 shows the per-system breakdown and Figure 3a (TC) the overall breakdown. Since a few bugs are triggered by more than one type of timing conditions (§3.3), the sum of numbers in Table 2 is slightly larger than the total number of DC bugs.

Message Timing Bugs. The timing conditions can be abstracted to two categories:

a. *Order violation* (44% in Table 2) means a DC bug manifests whenever a message comes earlier (later) than another event, which is another message or a local computation, but not when the message comes later (earlier).

b. *Atomicity violation* (20% in Table 2) means a DC bug manifests whenever a message comes in the middle of a set of events, which is a local computation or global communication, but not when the message comes either before or after the events.

LC and DC bugs are similar in that their timing conditions can both be abstracted into the above two types. However, the subjects in these conditions are different: shared memory accesses in LC and message deliveries in DC. The ratio between order violation and atomicity violation bugs are also different: previous study of LC bugs showed that atomicity violations are much more common than order violations in practice [45]; our study of DC bugs shows that this relationship does not apply or even gets reversed in several representative distributed systems.

	Ordering	Atomicity	Fault	Reboot
CA	4	4	6	5
HB	13	9	8	1
MR	25	4	5	3
ZK	4	8	7	5
All	46	25	26	14

Table 2. #DC bugs triggered by timing conditions (§3.1). *The total is more than 104 because some bugs require more than one triggering condition. More specifically, 46 bugs (44%) are caused* only *by ordering violations, 21 bugs (20%)* only *by atomicity violations, and 4 bugs (4%) by multiple timing conditions (as also shown in Figure 3a).*

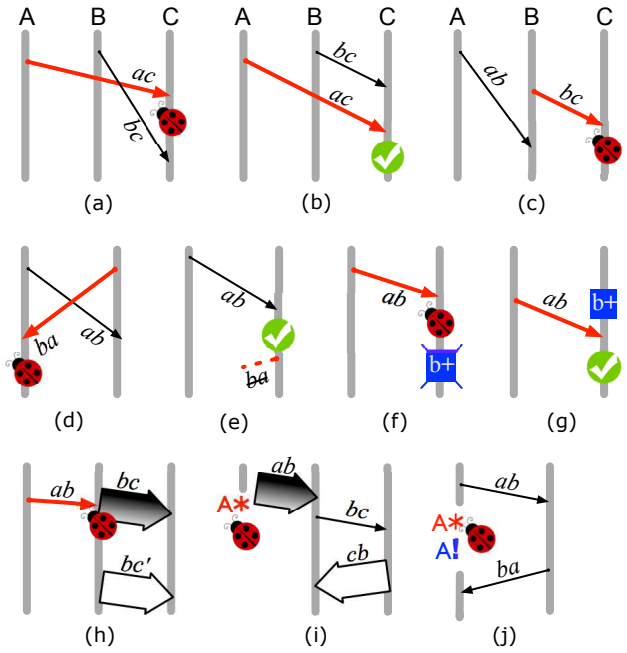

Figure 1. Triggering patterns (§3.1). *The three vertical lines represent the timeline of nodes A, B and C. An arrow with* `xy` *label implies a message from X to Y. A square box with label* `x+` *implies a local state-modifying computation at node X. A thick arrow implies a set of messages performing an atomic operation.* `X*` *and* `X!` *implies a crash and reboot at node X respectively (§2.6). All figures are discussed in §3.1*

An order violation can originate from a race between two messages (*message-message race*) at one node. The race can happen between two message arrivals. For example, Figure 1a illustrates ac/bc race at node C in m3274. Specifically, B_{RM} sends to C_{NM} a task-init message (bc_{init}), and soon afterwards, A_{AM} sends to C_{NM} a task-kill preemption message (ac_{kill}), however ac_{kill} arrives *before* bc_{init} and thus is incorrectly ignored by C. The bug would not manifest if ac_{kill} arrives *after* bc_{init} (Figure 1b). Message-message race can also happen between a message arrival and a message sending. For example, the ab/bc race in Figure 1c depicts h5780. In this bug, B_{RS} sends to C_{Master} a cluster-join request (bc_{join}) unexpectedly *before* a security-key message (ab_{key}) from A_{ZK} arrives at B, causing the initialization to abort.

Interestingly, message-message race can also occur concurrently across two nodes. For example, Figure 1d illustrates ab/ba race crisscrossing two nodes A and B in m5358. Specifically, A_{AM} sends ab_{kill} to a backup speculative task at B_{NM} because the job has completed, but concurrently the backup task at B sends $ba_{complete}$ to A, creating a double-complete exception at A. If ab_{kill} arrives early at B, *ba* will not exist and the bug will not manifest (Figure 1e).

An order violation can also originate from a race between a message and a local computation (*message-compute race*). For example, Figure 1f illustrates $ab/b+$ race in m4157. First, B_{AM} was informed that a task has finished and B plans to close the job and remove its local temporary files ($b+$). However, just *before* $b+$, A_{RM} sends to B a kill message (ab) and hence the files are never removed, eventually creating space issues. To prevent the failure, the kill message has to arrive after the local cleanup (Figure 1g).

An atomicity violation, as defined above, originates when a message arrives in the middle of a supposedly-atomic local computation or global communication. For example, Figure 1h illustrates m5009. When B_{NM} is in the middle of a commit transaction, transferring task output data (bc) to C_{HDFS}, A_{RM} sends a kill preemption message (ab) to B, preempting the task without resetting commit states on C. The system is never able to finish the commit — when B later reruns the task and tries to commit to C (bc'), C throws a double-commit exception. This failure would not happen if the kill message (ab) comes before or after the commit transaction (bc).

Fault and Reboot Timing Bugs. Fault and reboot timing bugs (32% in Table 2) manifest when faults and/or reboots occur at specific global states S_i; the bugs do not manifest if the faults and reboots happen at different global states S_j.

Figure 1i illustrates a fault-timing bug in m3858. Here, A_{NM1} is sending a task's output to B_{AM} (ab) but A crashes in the middle (A*) leaving the output half-sent. The system is then unable to recover from this untimely crash — B detects the fault and reruns the task at C_{NM2} (via bc) and later when C re-sends the output (cb), B throws an exception. This bug would not manifest, if the crash (A*) happens before/after the output transfer (ab).

Figure 1j depicts a reboot-timing bug in m3186. Here, A_{RM} sends a job (ab) to B_{AM} and while B is executing the job, A crashes and reboots (A*, A!) losing all its in-memory job description. Later, B sends a job-commit message (ba) but A throws an exception because A does not have the job information. The bug would not manifest if A reboots later: if A is still down when B sends ba_{commit} message, B will realize the crash and cancel the job before A reboots and A will repeat the entire job assignment correctly.

z1264: (1) *Follower F crashed* in the past, (2) *F reboots* and joins the cluster, (3) Leader L sync data with F and send snapshot, (4) *In the middle of step 3-6*, client updates data with Tx-#15; L forwards the update to F, (5) F applies the update in memory only, due to a concurrent sync, (6) L tells F syncing is finished, (7) Client updates data with Tx-#16; F writes update to disk correctly, (8) *F crashes*, (9) *F reboots* and joins the cluster again, (10) L sync data with F, but this time L sends only "diff" starting with Tx-#17 (11) F permanently *loses data* from Tx-#15, inconsistent with L and other followers!

Figure 2. A DC bug in ZooKeeper.

Message-Fault Bugs. Four DC bugs are caused by a combination of messages and faults. For example, in Figure 2, a message (step 4) arrives in the middle of some atomic operation (step 3-6). This message atomicity violation leads to an error that further requires a fault timing (step 8) to become an externally visible failure.

Finding #1: DC bugs are triggered mostly by *untimely messages* (64% in Table 2) and sometimes by *untimely faults/reboots* (32%), and occasionally by a *combination* of both (4%). Among untimely messages, two thirds commit order violations due to message-message or message-computation race on the node they arrive; the others commit atomicity violations.

3.2 Input Preconditions (IP)

The previous section presents simple timing conditions that can be understood in few simple steps. In practice, many of the conditions happen "deep" in system execution. In other words, the triggering path is caused by complex input preconditions (IP) such as faults, reboots, and multiple protocols. Let's use the same example in Figure 2. First, a fault and a reboot (step 1-2) and a client request (step 4) must happen to create a path to the message atomicity violation (step 4 interfering with step 3-6). Second, conflicting messages from two different protocols (ZAB and NodeJoin initiated in step 2 and 4) have to follow specific bug-triggering timing conditions. Even after the atomicity violation (after step 6), the bug is not guaranteed to lead to any error yet (*i.e.*, a benign race). Finally, the follower experiences an untimely fault (step 8), such that after it reboots (step 9), a global replica-inconsistency error will happen (step 11). Put it in a reverse way, before step 8, the global state is S_i and S_i+crash→error, and the only way for the system to reach S_i is from complex preconditions such as a fault, a reboot, and some foreground and background protocols.

Statistically, Figure 3b (IP-FLT) shows that 63% of DC bugs must have at least one fault. In more detail, Figure 3c-e (IP-TO, IP-CR, IP-RB) shows the percentage of issues that require timeouts, crashes and reboots respectively, including how many instances of such faults must be there; the rest is other faults such as disk errors (not shown).

Figure 3f (IP-PR) shows how many "protocol initiations" mentioned in the bug description. For example, if the system needs to perform one execution of background protocol and

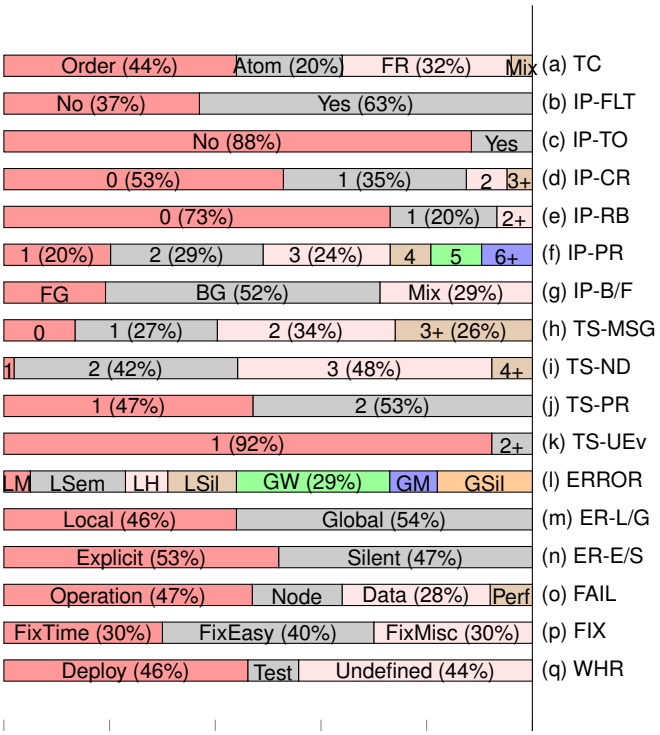

Figure 3. Statistical overview of TaxDC. *Timing Conditions (TC) is discussed in §3.1, Input Preconditions (IP) in §3.2, Triggering Scope (TS) in §3.3, Errors (ER) in §4.1, Failures (FAIL) in §4.2, Fixes (FIX) in §5, and Where Found (WHR) in §7.*

also three concurrent calls to the `write` protocol, then we label it with four protocol initiations. Up to 3 protocol initiations covers three quarters of DC bugs. When we count the number of *unique* protocols involved in all the bugs we study, we record 10 Cassandra, 13 HBase, 10 MapReduce, 6 ZooKeeper unique protocols, or 39 protocols in total. This again highlights the complexity of fully complete systems. Figure 3g (IP-B/F) shows our categorization of protocols that are concurrently running into foreground only, background only, and foreground-background (mix) categories. More than three quarters of the bugs involve some background protocols and about a quarter involves a mix of foreground and background protocols.

Finding #2: Many DC bugs need *complex input preconditions*, such as faults (63% in Figure 3b), multiple protocols (80% in Figure 3f), and background protocols (81% in Figure 3g) .

3.3 Triggering Scope (TS)

We now analyze the triggering scope (TS), which is a complexity measure of DC-bug timing conditions. We use four metrics to measure the scope: message count (TS-MSG), node (TS-ND), protocol (TS-PR), and untimely event (TS-UEv) counts as shown in Figure 3h-k. This statistic is important with respect to the scalability of model checking, bug detection and failure diagnostic tools (§8.2-8.5).

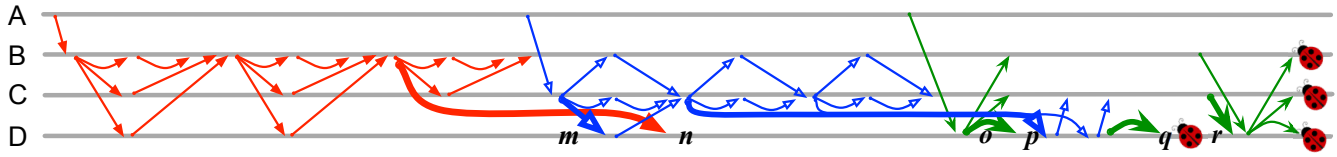

Figure 4. A Cassandra's Paxos bug. *In c6023, three key-value updates (different arrow types) concurrently execute the Paxos protocol on four nodes (we simplify from the actual six nodes). The bug requires three message-message race conditions: (1)* m *arrives before* n*, (2)* o *before* p*, and (3)* q *before* r*, which collectively makes D corrupt the data and propagate the corruption to all replicas after the last broadcast. Note that the bug would not surface if any of the conditions did not happen. It took us one full day to study this bug.*

Message count implies the minimum number of messages involved in E as defined in the beginning of section 3. Figure 3h (TS-MSG) shows that one or two triggering messages are the most common, with 7 messages as the maximum. Informally, zero implies fault timing bugs without any message-related races, one implies message-compute race, two implies message-message as in Figure 1a, and three implies a scenario such as *ac*/(*ab*-*bc*) race where *ab* and *ac* are concurrent or non-blocking message sending operations.

The node and protocol scopes present how many nodes and protocols are involved within the message scope. Figure 3i-j (TS-ND and TS-PR) shows that the scale of node and protocol triggering scope is also small, mostly two or three nodes and one or two protocols.

The untimely events count implies the total number of order violations, atomicity violations, untimely faults and reboots in the triggering timing condition of a bug. Figure 3k (TS-UEv) shows that only eight bugs require more than one untimely events. Four of them are message-fault bugs, each requiring one untimely message and one untimely fault to trigger (*e.g.*, step 4 and 8 in Figure 2). Three are fault-reboot timing bugs, each requiring one untimely fault and one untimely reboot. The last one is c6023, shown in Figure 4, requiring three message-message order violations to happen.

Finding #3: The *timing conditions* of most DC bugs only involve *one to three* messages, nodes, and protocols (>90% in Figure 3h-j). Most DC bugs are mostly triggered by only *one* untimely event (92% in Figure 3k).

4. Errors and Failures

"... we [prefer] debugging crashes instead of hard-to-track hanging jobs." — m3634

4.1 Error Symptoms

From the triggering conditions, we then scrutinize the *first* error that happens immediately after. First errors are the pivotal point that bridges the triggering and error-propagation process. Identifying first errors help failure diagnosis get closer to disclosing bug triggering and root causes and help bug detection get closer to accurately predict failures (§8).

	Local Errors				Global Errors		
	Mem	Sem	Hang	Sil	Wrong	Miss	Sil
CA	2	0	0	4	3	3	7
HB	1	2	1	2	15	3	6
MR	2	13	7	4	14	4	0
ZK	0	6	2	5	1	0	5
All	5	21	10	15	33	10	18

Table 3. First error symptoms of DC bugs (§4.1). *Some bugs cause multiple concurrent first errors.*

We categorize first errors into *local* errors and *global* errors, based on whether they can be observed from the triggering node N_T alone. Here, N_T is the node where triggering ends. It is the receiver node of untimely messages (*e.g.*, node C in Figure 1a) or the node with untimely fault (*e.g.*, node A in Figure 1i). For each error, we also check whether it is an *explicit* or *silent* error. Table 3 and Figure 3l (ERR) show the per-system and overall breakdowns respectively. Some MapReduce bugs caused multiple concurrent first errors of different types.

First, DC bugs can manifest into both local explicit errors and local silent errors. The former includes *memory exceptions* such as null-pointer exceptions (5% in Table 3) and *semantic errors* such as wrong state-machine transition exceptions thrown by the software (19%). Local silent errors include *hangs*, such as forever waiting for certain states to change or certain messages to arrive which are typically observed implicitly by users (9%), and *local silent* state corruption, such as half-cleaned temporary files (13%).

When local error is non-obvious in N_T, we analyze if the error is observable in other nodes communicating with N_T. Many DC bugs manifest into explicit global errors through *wrong messages* (29% in Table 3). Specifically, the communicating node receives an incorrect message from N_T, and throws an exception during the message handling. However, a few DC bugs still lead to silent global errors. These include *missing messages*, where N_T never sends a reply that the communicating node is waiting for in the absence of timeout (9%), and *global silent* state corruption such as replica inconsistencies between N_T and the other nodes (16%).

Finding #4: *Local* and *global* first errors are about equally common; 46% vs. 54% in Figure 3m (ER-L/G). About half of the DC bugs generate *explicit* first errors (53%), including local exceptions and global wrong messages, and the remaining DC bugs lead to *silent* errors (47%), as shown in Figure 3n (ER-E/S). Some of them immediately lead to hangs in the triggering node N_T (9%) or a node communicating with N_T (9%).

4.2 Failure Symptoms

Figure 3o (FAIL) shows that errors from DC bugs will eventually lead to a wide range of fatal failures including node downtimes (17%), data loss/corruption/inconsistencies (28%), operation failures (47%), and performance degradation (8%). A node downtime happens when the node either crashes or hangs (*i.e.*, it may still be heartbeating). It happens to both master/leader nodes and worker/follower nodes in our study. Data-related failures and performance problems are an artifact of incorrect state logic induced from DC bugs. For example, in HBase, concurrent region updates and log splittings can cause data loss. In Cassandra, some dead nodes are incorrectly listed as alive causing unnecessary data movement that degrades performance. Node downtimes and data-related failures could also cause some operations to fail. To avoid double counting, we consider a bug as causing operation failures only when it does not cause node downtimes or data-related failures.

5. Fixes

"We have already found and fix many cases ... however it seems exist many other cases." — h6147

We next analyze bug patches to understand developers' fix strategies. In general, we find that DC bugs can be fixed by either disabling the triggering timing or changing the system's handling to that timing (*fix timing* vs. *fix handling*). The former prevents concurrency with extra synchronization and the latter allows concurrency by handling untimely events properly. Since message timing bugs are fixed quite differently from fault timing bugs, we separate them below.

5.1 Message Timing Bug Fixes

The left half of Table 4 shows that only one fifth of message timing bugs are fixed by disabling the triggering timing, through either global or local synchronization. Only a couple of bugs are fixed through extra *global synchronization*, mainly due to its complexity and communication cost. For example, to prevent a triggering pattern $b+/ab$ in Figure 1f, m5465's fix *adds* a monitor on A_{RM} to wait for ba_{done} message from B_{AM} after B finishes with its local computation ($b+$); the result is *b+-ba-ab* global serialization. More often, the buggy timing is disabled through *local synchronization*, such as re-ordering message sending operations within a single node. For example, h5780's fix for ab/bc race in Figure 1c forces the sending of *bc* request at B to wait for the receipt of *ab*; the result is *ab-bc* local serialization at B.

	Fix Timing		Fix Handling			
	Glob	Loc	Ret	Ign	Acc	Misc
CA	0	0	0	1	3	4
HB	2	7	2	1	7	3
MR	2	8	2	7	8	3
ZK	0	4	0	3	0	1
All	4	19	4	12	18	11

Table 4. Fix strategies for message timing bugs (§5.1). *Some bugs require more than one fix strategy.*

The right half of Table 4 shows that fix handling is more popular. Fix handling fortunately can be simple; many fixes do *not* introduce brand-new computation logic into the system, which can be done in three ways. First, the fix can handle the untimely message by simply *retrying* it at a later time (as opposed to ignoring or accepting it incorrectly). For example, to handle bc/ac race in Figure 1a, m3274 retries the unexpectedly-early ac_{kill} message at a later time, right after the to-be-killed task starts. Second, the fix can simply *ignore* the message (as opposed to accepting it incorrectly). For example, to handle ab/ba race in Figure 1d, m5358 simply ignores the unexpectedly-late ba_{finish} message that arrives after A_{AM} sends an ab_{kill} message. Finally, the patch can simply *accept* the untimely message by *re-using* existing handlers (as opposed to ignoring it or throwing an error). For example, m2995's fix changes the node AM to accept an unexpectedly-early expiration message using an existing handler that was originally designed to accept the same message at a later state of AM. m5198's fix handles the atomicity violation by using an existing handler and simply cancels the atomicity violating local operation. The rest of the fix-handling cases require new computation logic to fix bugs.

5.2 Fault/Reboot Timing Bug Fixes

Table 5 summarizes fix strategies for fault/reboot timing bugs. Unlike message timing, only rare bugs can be fixed by controlling the triggering timing either globally or locally (*e.g.*, by controlling the timing of the fault recovery actions). A prime example is an HBase cluster-wide restart scenario (h3596). Here, as A shuts down earlier, B assumes responsibility of A's regions (via a region-takeover recovery protocol), but soon B shuts down as well with the regions still locked in ZooKeeper and the takeover cannot be resumed after restart. The patch simply adds a delay before a node starts region takeover so that it will likely get forced down before the takeover starts.

For the majority of fault timing bugs, their patches conduct two tasks: (1) detect the local/global state inconsistency caused by the fault and (2) repair/recover the inconsistency. The former is accomplished through timeouts, additional message exchanges, or others (omitted from Table 5).

	Fix Timing		Fix Handling			
	G	L	Detect		Recover	
			TO	Msg	Canc	Misc
CA	1	0	3	2	4	6
HB	0	1	3	1	6	1
MR	2	1	1	1	2	1
ZK	0	3	0	1	1	7
All	3	5	7	5	13	15

Table 5. Fix strategies for fault/reboot timing bugs (§5.2). *Some bugs require more than one fix strategy.*

The latter can be achieved by simply canceling operations or adding new computation logic.

Finding #5: A *small number of fix strategies* have fixed most DC bugs. A few DC bugs are fixed by *disabling* the triggering timing (30% in Figure 3p), occasionally through extra messages and mostly through local operation re-orderings. Most DC bugs are fixed by better handling the triggering timing, most of which do not introduce new computation logic — they *ignore* or *delay* messages, *re-use* existing handlers, and *cancel* computation (40%).

6. Root Causes

"This has become quite messy, we didn't foresee some of this during design, sigh." — m4819

It is difficult to know for sure why many DC-bug triggering conditions were not anticipated by the developers (*i.e.*, the root causes). In this section, we postulate some possible and common misbeliefs behind DC bugs.

"One hop is faster than two hop." Some bugs manifest under scenario *bc*/(*ba-ac*), similar to Figure 1a. Developers may assume that *bc* (one hop) should arrive earlier than *ba-ac* (two hops), but *ac* can arrive earlier and hit a DC bug.

"No hop is faster than one hop." Some bugs manifest under scenario *ba*-(*b+*/*ab*), similar to Figure 1f. Developers may incorrectly expect *b+* (local computation with no hop) to always finish before *ab* arrives (one hop).

"Atomic blocks cannot be broken." Developers might believe that "atomic" blocks (local or global transactions) can only be broken unintentionally by some faults such as crashes. However, we see a few cases where atomic blocks are broken inadvertently by the system itself, specifically via untimely arrival of kill/preemption messages in the middle of an atomic block. More often, the system does not record this interruption and thus unconsciously leaves state changes half way. Contrary, in fault-induced interruption, some fault recovery protocol typically will handle it.

"Interactions between multiple protocols seem to be safe." In common cases, multiple protocols rarely interact, and even when they do, non-deterministic DC bugs might not surface. This can be unwittingly treated as normally safe, but does not mean completely safe.

"Enough states are maintained." Untimely events can unexpectedly corrupt system states and when this happens the system does not have enough information to recollect what had happened in the past, as not all event history is logged. We observe that some fixes add new in-memory/on-disk state variables to handle untimely message and fault timings.

Finding #6: Many DC bugs are related with a few common misconceptions that are unique to distributed systems.

7. Other Statistics

"Great catch, Sid! Apologies for missing the race condition." — m4099

We now present other quantitative findings not included in previous discussions. We attempted to measure the complexity of DC bugs using four metrics: (a) the number of "re-enumerated steps" as informally defined in §2.4, (b) the patch LOC including new test suites for the corresponding bug, (c) the time to resolve (TTR), and (d) the number of discussion comments between the bug submitter and developers. The 25th percentile, median, and 75th percentile for the four metrics are (a) 7, 9, and 11 steps, (b) 44, 172, and 776 LOC, (c) 4, 14, and 48 days to resolve, (d) 12, 18, and 33 comments.

In terms of where the bugs were found, Figure 3r (WHR) highlights that 46% were found in deployment and 10% from failed unit tests. The rest, 44%, are not defined (could be manually found or from deployment). Some DC bugs were reported from large-scale deployments such as executions of thousands of tasks on hundreds of machines.

Due to space constraints, we do not provide cross-cutting analyses (*e.g.*, how many bugs have >2 protocols and >2 crashes with patch >100 LOC and were found in deployment). However, future TaxDC users can easily do so with all the rich statistics stored in TaxDC database.

8. Lessons Learned

"This issue, and the other related issues ... makes me very nervous about all the state combinations distributed between [ZooKeeper and many HBase components]. After this is done, do you think we can come up with a simpler design? I do not have any particular idea, so just spitballing here." — h6060

We now discuss the lessons learned, implications to existing tools and the opportunities for new research in combating DC bugs. Although many high-level directions can be

adopted from work on LC bugs, there are many interesting challenges and opportunities unique to DC bugs.

8.1 Fault Paths and Multi-Protocol Interactions

Individual protocols tend to be robust in general. Only 18 DC bugs occur in *individual* protocols *without* any input fault condition; only 8 of them are in foreground protocols. On the other hand, a large majority of DC bugs happen due to concurrent executions of multiple protocols and/or different fault timings (Finding **#2**). This has a tremendous implication to input testing: *all types of verification, testing, and analysis approaches must consider fault injections and multiple protocols as input conditions.* Although recent work has paid attention to this [24, 35, 67], we emphasize that all forms of faults (§2.6) must be exercised.

8.2 Distributed Systems Model Checkers

Assuming the necessary input conditions are exercised, the next question is: can we test different event re-orderings to hit the triggering timing (§3.1)? This is the job of distributed system model checkers (dmck), which are gaining popularity recently [26, 37, 38, 59, 65]. Dmck works by intercepting distributed events and permuting their ordering. The more events included, the more scalability issues will arise due to state-space explosion. To date, *no dmck completely controls the timings of all necessary events* that might contribute to the triggering timing (Finding **#1**). MaceMC [37] only reorders messages and network disconnections. MoDist [26] exercises timeouts and Demeter [26] intercepts messages and local computation but they do not explore different timing of multiple crashes and reboots. SAMC [38] exercises multiple faults but does not support timeout and thread controls. Also, none of the above include storage faults or timing issues [27]. Therefore, continued research on scalable exploration algorithms is needed, specifically when *all* the necessary events need to be controlled. This could be helped by DC bugs' triggering scope characteristics (Finding **#3**), just like that in LC model checkers [48].

8.3 Domain-Specific Specifications

Now, assuming the necessary events are controlled, the next question is: do we have the specification to judge the manifestation of a bug? This is a plague for many tools. For example, Demeter does not find new bugs [26] and SAMC [38] finds two new bugs. Conversations with the authors suggest that their target systems do not deploy detailed specifications, and thus some bugs are left uncaught. Deploying generic "textbook" specifications (*e.g.*, "only one leader exists") does not help as they could lead to false positives (*e.g.*, ZooKeeper allows two leaders at a single point in time). Many research papers on specifications only deploy few of them [24, 42, 55]. Developers also bemoan the hard-to-debug fail-silent problems m3634 and prefer to see easier-to-debug fail-stop bugs.

On the positive side, 53% of DC bugs lead to explicit first errors (Finding **#4**), implying that sanity checks already in software can be harnessed as specifications (more in §8.4). On the other side, compared to single-machine systems, distributed systems are much more capable of masking errors. Therefore, these error specifications have to be used with caution to avoid false positives. Furthermore, 47% of DC bugs lead to silent first errors (Finding **#4**). Many of them proceed to "silent failures", such as data loss, node hangs, etc. Even if they become explicit errors later, these explicit errors could be far away from the initial triggering conditions (*e.g.*, Figure 4). In short, *no matter how sophisticated the tools are, they are ineffective without accurate specifications*. This motivates the creation or inference of local specifications that can show early errors or symptoms of DC bugs.

8.4 Bug Detection Tools

We now discuss bug detection tools, which are unfortunately rare for DC bugs, although very popular for LC bugs [8, 20, 30, 46, 48, 56]. Bug detection tools look for bugs that match specific patterns. They cannot provide bug-free proof, but can be efficient in discovering bugs when guided by the right patterns. Our study provides guidance and patterns that can be exploited by future DC bug detection.

Generic detection framework. Finding **#1** implies that detecting DC bugs, particularly message-timing DC bugs, should focus on two key tasks: (1) obtaining timing specifications, including order and atomicity specifications among messages and computation; and (2) detecting violations to these specifications through dynamic or static analysis.

Invariant-guided detection. Likely program invariants can be learned from program behaviors, and used as specifications in bug detection [18, 19, 46]. The key challenge is to design simple and suitable invariant templates. For example, "function F_1 should always follow F_2" is a useful template for API-related semantic bugs [18]; "the atomicity of accesses a_1 and a_2 should never be violated" is effective for LC bugs [46]. Finding **#1** about triggering timing and Finding **#4** about error patterns provide empirical evidence that these templates can be effective for DC bugs: "message bc should arrive at C before message ac (ca) arrives (leaves)"; "message ab should never arrive in the middle of event e on node B"; and "message ab should always be replied".

Misconception-guided bug detection. Knowing programmers' misconceptions can help bug detectors focus on specifications likely to be violated. LC bug researchers have leveraged misconceptions such as *"two near-by reads of the same variable should return the same value"* [46] and *"a condition checked to be true should remain true when used"* [53]. Finding **#6** reveals that DC-unique common misconceptions, such as *"a single hop is faster than double hops"*, *"local computation is faster than one-hop message"*, *"atomic blocks cannot be broken"* can help DC bug detection.

Error-guided bug detection. Finding **#4** shows that many DC bugs lead to explicit local/global errors, which implies that timing specifications for many DC bugs can be inferred backward based on explicit errors. For example, program analysis may reveal that a state-machine exception e will arise whenever C receives message ac before bc, which provides a timing specification (ac arrives before bc) whose violation leads to a *local error*; or, the analysis may reveal that exception e arises whenever node B receives a message cb from node C and C only sends cb when ac arrives at C before bc, which provides a timing specification whose violation leads to a *wrong-message global error*; and so on.

Software testing. Testing takes a quarter of all software development resources, and is crucial in exposing bugs before code release. Although many testing techniques have been proposed for LC bugs [9, 51, 57], there have been few for DC bugs [58]. Finding **#2** implies that test input design has to consider faults, concurrent protocols, and background protocols. Finding **#3** implies that *pairwise testing*, which targets every pair of message ordering, every pair of protocol interaction, and so on, will work much more effectively than *all combination testing*, which exercises all possible total orders and interactions of all messages and all protocols. For example, a large number of DC bugs (Figure 3d-f) can be found with inputs of at most two protocols, crashes and reboots.

8.5 Failure Diagnosis

Given failure symptoms, distributed systems developers have to reason about many nodes to figure out the triggering and root cause of a failure. Our study provides guidance to this challenging process of failure diagnosis.

Slicing/dependence analysis. Identifying which instructions can affect the outcome of an instruction i is a widely used debugging technique for deterministic sequential bugs. However, it cannot scale to the whole distributed systems, and hence is rarely used. Finding **#3** indicates that most DC bugs have deterministic error propagation; Finding **#4** shows that many DC bugs have their errors propagate through missing or wrong messages. Therefore, per-node dependence analysis that can quickly identify whether the generation of a local error depends on any incoming messages would help DC bug failure diagnosis to get closer and closer to where the triggering events happen.

Error logging. Error logging is crucial in failure diagnosis. If the first error of a DC bug is an explicit local error, the error log can help developers quickly identify the triggering node and focus their diagnosis on one node. Finding **#4** unfortunately shows that only 23% of DC bugs lead to explicit local errors. This finding motivates future tool to help make more DC bugs lead to explicit local errors.

Statistical debugging. Comparing success-run traces with failure-run traces can help identify failure predictors for semantic bugs [40] and concurrency bugs [33] in single-machine software. The key design question is what type of program properties should be compared between failure and success runs. For example, branch outcomes are compared for diagnosing semantic bugs but not for LC bugs. Finding **#1** and **#3** about triggering timing conditions provide guidance for applying this approach for DC bugs. We can collect all message sending/arrival time at runtime, and then find rare event orderings that lead to failures by contrasting them with common "healthy" orderings (*e.g.*, Figure 1b happens 99.99% of the time while Figure 1a happens 0.01% of the time). Of course, there are challenges. Finding **#2** and **#3** show that many DC bugs come from the interactions of many protocols. Thus, it is not sufficient to only log a chain of messages originated from the same request, a common practice in request logging [13]. Furthermore, some DC bugs are triggered by message-computation ordering. Therefore, logging messages alone is not sufficient.

Record and Replay. Debugging LC concurrency bugs with record and deterministic replay is a popular approach [52, 61]. However, such an approach has not permeated practices in distributed systems debugging. A ZooKeeper developer pointed us to a fresh DC bug that causes a whole-cluster outage but has not been fixed for months because the deployment logs do not record enough information to replay the bug (z2172). There has been 9 back-and-forth log changes and attachments with 72 discussion comments between the bug submitter and the developers. More studies are needed to understand the gap between record-replay challenges in practice and the current state of the art [23, 43].

8.6 Failure Prevention and Fixing

Runtime Prevention. The manifestation of concurrency bugs can sometimes be prevented by injecting delays at runtime. This technique has been successfully deployed to prevent LC bugs based on their timing conditions [36, 47, 68]. Finding **#1** shows that many DC bugs are triggered by untimely messages and hence can potentially be prevented this way. For example, none of the bugs shown in Figure 1a–h would happen if we delay a message arrival/sending or local computation. Of course, different from LC bugs, some of these delays have to rely on a network interposition layer; similar with LC bugs, some delays may lead to hangs, and hence cannot be adopted.

Bug Fixing. Recent work automatically fixes LC bugs by inserting lock/unlock or signal/wait to prohibit buggy timing [34, 41, 62]. Finding **#5** shows that the same philosophy is promising for 30% of studied DC bugs. Our study shows that this approach has to be tweaked to focus on using global messages (*e.g.*, ACKs) or local operation re-ordering, instead of lock or signal, to fix DC bugs. Finding **#5** indicates that 40% of those DC bugs are fixed by shifting message handlers, ignoring messages, and canceling computa-

h9095: (1) RS successfully OPENED region R, (2) RS notifies ZK that region R is OPENED, (3) ZK continues region R state msg to Master, (4) Master starts processing OPENED msg, (5) Meanwhile RS CLOSED region R (asked by Client), (6) RS notifies ZK that region R is CLOSED, (7) Master asks ZK to delete znode for region R, *concurrently racing with step 6!*, (8) ZK deletes region R's znode, (9) Master never assigns region R to any RS. *R becomes an orphan!*

Figure 5. Race of HBase's messages to ZooKeeper.

tion, without adding new computation logic. This presents a unique opportunity for developing new and more aggressive fixing techniques.

8.7 Distributed Transactions

In the middle of our study, we ask ourselves: if DC bugs can be theoretically solved by distributed transactions, why doesn't such technique eliminate DC bugs in practice? Our answers are: first, the *actual implementations of theoretically-proven distributed transactions are not always correct* (as also alluded in other work [10, 49]). For example, new DC bugs continue to surface in complex distributed transactions such as ZooKeeper's ZAB and Cassandra's Paxos as they are continuously modified. Second, *distributed transactions are only a subset of a full complete system*. A prime example is the use of ZooKeeper in HBase for coordinating and sharing states between HBase masters and region servers. Although ZooKeeper provides linearization of updates, HBase must handle its concurrent operations to ZooKeeper, for example, step 6 and 7 in Figure 5; there are many other similar examples.Put simply, there are many protocols that do not use distributed transactions, instead they use domain-specific finite state machines, which should be tested more heavily.

Another approach to eliminate non-deterministic bugs in distributed protocols is by building deterministic distributed systems. However, the technique is still in its infancy, at least in terms of the impact to performance (*e.g.*, an order of magnitude of overhead [31]).

8.8 Verifiable Frameworks

Recently there is a growing work on new programming language frameworks for building verifiable distributed systems [16, 28, 63], but they typically focus on the main protocols and not the full system including the background protocols. One major challenge is that just for the basic read and write protocols, the length of the proofs can reach thousands of lines of code, potentially larger than the protocol implementation. Unfortunately, our study shows that the complex interaction between foreground and background protocols can lead to DC bugs. Therefore, for complete real-world systems, verification of the entire set of the protocols is needed.

8.9 LC bugs vs. DC bugs

There are clearly similarities between LC bugs and DC bugs, as, by definition, they are both timing-related non-deterministic bugs. Many DC bugs contain LC components: untimely messages may lead to unsynchronized accesses from multiple threads or multiple event-handlers [30, 54] in a single machine. It is probably not a surprise that atomicity violations and order violations are two dominant triggering timing conditions for both LC and DC bugs (Finding **#1**). Our observation of the small triggering scope of most DC bugs (Finding **#3**) is similar with that for LC bugs, which may be related to the nature of the bug sets — more complicated bugs may be more difficult to fix, and hence less likely to be included in empirical studies.

There are also many differences between LC bugs and DC bugs, as they originate from different programming paradigms and execution environments. For example, order violations are much more common in DC bugs than those in LC bugs (Finding **#1**); faults and reboots are much more common in DC bugs than those in LC bugs (Finding **#2**); the diagnosis of many DC bugs will have to reason beyond one node, clearly different from that of LC bugs (Finding **#4**); the fix strategies for DC bugs are very different from those of LC bugs, because enforcing global synchronization is difficult (Finding **#5**).

9. Conclude

DC Bugs impact the reliability, availability, and performance of real-world distributed systems. Even with all the redundancy and fault-recovery mechanisms deployed in today's systems, DC bugs make the software as the single point of failure. Combating DC bugs will have a profound impact to future software systems as more organizations are building more distributed system layers on farms of machines and services in this era of cloud computing. The gap between the theory and practice of distributed systems reliability should continue to be narrowed. We hope our work will commence more interdisciplinary actions from diverse researchers and practitioners in the areas of concurrency, fault tolerance and distributed systems to combat DC bugs together.

10. Acknowledgments

We thank the anonymous reviewers for their tremendous feedback and comments. We also would like to thank Chen Tian of Huawei and Yohanes Surya and Nina Sugiarti of Surya University for their continuing supports. This material is based upon work supported by the NSF (grant Nos. CCF-1336580, CNS-1350499, CCF-1439091, CNS-1514256, IIS-1546543) and generous supports from Huawei and Alfred P. Sloan Foundation. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] http://www.freepastry.org/.

[2] http://ucare.cs.uchicago.edu/projects/cbs/.

[3] Apache Cassandra. http://cassandra.apache.org.

[4] Apache Hadoop. http://hadoop.apache.org.

[5] Apache Hadoop NextGen MapReduce (YARN). https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[6] Apache HBase. http://hbase.apache.org.

[7] Apache ZooKeeper. http://zookeeper.apache.org.

[8] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional Detection of Data Races. In *PLDI*, 2010.

[9] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*, 2010.

[10] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06*.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI '06*.

[12] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *SOSP '01*.

[13] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end performance analysis of large-scale Internet services. In *OSDI '14*.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04*.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP '07*.

[16] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe Asynchronous Event-Driven Programming. In *PLDI '13*.

[17] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *SoCC '13*.

[18] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP*, 2001.

[19] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly Detecting Relevant Program Invariants. In *ICSE*, 2000.

[20] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[21] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A Study of the Internal and External Effects of Concurrency Bugs. In *DSN*, 2010.

[22] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *NSDI '07*.

[23] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay Debugging for Distributed Applications. In *USENIX ATC '06*.

[24] 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. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI '11*.

[25] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *SoCC '14*.

[26] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *SOSP '11*.

[27] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *FAST '16*.

[28] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *SOSP '15*.

[29] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI '11*.

[30] Chun-Hung Hsiao, Cristiano L. Pereira, Jie Yu, Gilles A. Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. Race Detection for Event-Driven Mobile Applications. In *PLDI*, 2014.

[31] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. DDOS: Taming Nondeterminism in Distributed Systems. In *ASPLOS '13*.

[32] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. RADBench: A Concurrency Bug Benchmark Suite. In *HotPar*, 2011.

[33] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation. In *OOPSLA*, 2010.

[34] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated Concurrency-Bug Fixing. In *OSDI*, 2012.

[35] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *OOPSLA '11*.

[36] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock Immunity: Enabling Systems To Defend Against Deadlocks. In *OSDI*, 2008.

[37] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI '07*.

[38] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *OSDI '14*.

[39] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A Characteristic Study on Failures of Production Distributed Data-Parallel Programs. In *ICSE*, 2013.

[40] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug Isolation via Remote Program Sampling. In *PLDI*, 2003.

[41] Peng Liu, Omer Tripp, and Charles Zhang. Grail: Context-Aware Fixing of Concurrency Bugs. In *FSE*, 2014.

[42] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI '08*.

[43] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI '07*.

[44] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *FAST '13*.

[45] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS '08*.

[46] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, 2006.

[47] Brandon Lucia and Luis Ceze. Cooperative Empirical Failure Avoidance for Multithreaded Programs. In *ASPLOS*, 2013.

[48] Madanlal Musuvathi and Shaz Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, 2007.

[49] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX ATC '14*.

[50] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *ASPLOS*, 2011.

[51] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Finding Places. In *ASPLOS*, 2009.

[52] Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu, Justin Gottschlich, Nima Honarmand, Nathan Dautenhahn, Samuel T. King, and Josep Torrellas. QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs. In *ISCA*, 2013.

[53] Shanxiang Qi, Abdullah A. Muzahid, Wonsun Ahn, and Josep Torrellas. Dynamically Detecting and Tolerating IF-Condition Data Races. In *HPCA*, 2014.

[54] Veselin Raychev, Martin T. Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *OOPSLA*, 2013.

[55] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI '06*.

[56] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM TOCS*, 1997.

[57] Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, 2008.

[58] Koushik Sen and Gul Agha. Automated Systematic Testing of Open Distributed Programs. In *FSE*, 2006.

[59] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *SSV '10*.

[60] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM '01*.

[61] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, 2011.

[62] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *OSDI*, 2008.

[63] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed System. In *PLDI*, 2015.

[64] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. Nondeterminism in MapReduce Considered Harmful? An Empirical Study on Non-commutative Aggregators in MapReduce Programs. In *ICSE '14*.

[65] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI '09*.

[66] Jie Yu. A collection of concurrency bugs. https://github.com/jieyu/concurrency-bugs.

[67] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *OSDI '14*.

[68] Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *ASPLOS*, 2013.