THE UNIVERSITY OF CHICAGO

UNEARTHING CONCURRENCY AND SCALABILITY BUGS IN CLOUD-SCALE DISTRIBUTED SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
TANAKORN LEESATAPORNWONGSA

CHICAGO, ILLINOIS
AUGUST 2017

To my family: father, mother, Louise, Fon, Nuch, and Nid

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.” — Brian Kernigham

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Acknowledgment here

ABSTRACT

In the era of cloud computing, users (normal and organization) move their data and computation from local machines and internal datacenters to cloud services, thus the services are expected to be 24/7 dependable. Cloud services must be accessible anytime and anywhere and not lose or corrupt users data, and scale as user base continues to grow. Unfortunately, guaranteeing cloud services' dependability is challenging because these cloud services are backed by large sophisticated distributed systems such as scalable data stores, data-parallel frameworks, and cluster management systems. Such cloud-scale distributed systems remain difficult to get right because they need to address data races among nodes, complex failures in commodity hardware, tremendous user requests, and much more. Addressing these cloud-specific challenges makes the systems getting more complex and new intricate bugs continue to create dependability problems.

This dissertation tries to answer a vital question of cloud dependability: "*how can we make cloud-scale distributed systems reaching ideal dependability?*" We try to answer this question by focusing on the problems of distributed concurrency bugs and scalability bugs. We focus on these two problems because they are novel issues that occur in cloud environment only and not many works addressing them.

Distributed concurrency bug (DC bug) is one of unsolved reliability problem in cloud systems. DC bugs are caused by non-deterministic order of distributed events such as message arrivals, machine crashes, and reboots. Cloud systems execute multiple complicated distributed protocols concurrently. The possible interleavings of the distributed events are beyond developer's anticipations and some interleavings might not be handled properly that can lead to catastrophic failures. To combat DC bugs, we make two contributions. First, we advance state of the art of distributed system model checking by introducing "*semantic-aware model checking" (SAMC)*. Distributed system model checkers (dmck) are used to test system reliability of real systems. Existing dmcks however rarely exercise multiple faults due to the state-space explosion problem, and thus do not address present reliability challenges of cloud systems in dealing with complex faults. SAMC

pushes the boundary of dmcks by introducing a white-box principle that takes simple semantic information of the target system and incorporates that knowledge into state-space reduction policies. We show that SAMC can find deep bugs one to two orders of magnitude faster compared to state-of-the-art techniques.

Second, we conduct a formal study on DC bugs to gain foundation knowledge for future DC bug research. We study 104 DC bugs from various widely-deployed cloud-scale distributed systems in many characteristics along several axes of analysis such as the triggering timing condition and input preconditions, error and failure symptoms, and fix strategies. And we present the first complete taxonomy of DC bugs, TaxDC.

And for the second aspect of system dependability, we focus on scalability bugs. Scale surpasses the limit of a single machine in meeting users' increasing demands for computing and storage. On the negative side, scale creates new development and deployment issues. Developers must ensure that their algorithms and protocol designs to be scalable. However, until real deployment takes place, unexpected bugs in the actual implementations are unforeseen. This new era of cloud-scale distributed systems has given birth to "scalability bugs", latent bugs that are scale-dependent, and only surface in large scale.

To address scalability bugs, we conduct a study on scalability bugs to understand how they manifest and what their root causes are and introduce SCK, a methodology that enables developers to *scale-check* distributed systems and find scalability bugs economically on one machine. SCK helps developers identify potential buggy code and allows developers to colocate a large number of nodes to test the potential buggy code without sacrificing accuracy. We remove a problem of hardware contentions (*i.e.*, CPU, memory, and thread) with four novel strategies, and we successfully integrate SCK to Cassandra, HDFS, Riak, and Voldemort. With SCK, we achieve a high colocation factor (500 nodes), and can reproduce ten scalability bugs and identify two new hidden bugs.

CHAPTER 1

INTRODUCTION

"*Cloud computing*" has been given many definitions from many companies and experts [28, 29, 30, 55]. These definitions are different in details, but they have some common characteristics; they are on-demand internet-based services that can scale to fit increasing users, and users pay only for their use. Cloud computing help users (from end users to organizational users) reduce the capital investment in hardware that is mostly underutilized [69] and help business moves faster [99]. We see a trend that users are moving their data and computation from local machines and in-house datacenters to the cloud [2, 8, 17, 19, 24, 26, 27, 98].

This trend makes client-side software get thinner and more heavily rely on the cloud services, thus the services are expected to be 24/7 dependable. Cloud services must be accessible anytime and anywhere and not lose or corrupt users data, and scale as user base continues to grow [36]. Unfulfilled dependability is costly. Some researchers estimate that 568 hours of downtime at 13 well-known cloud services since 2007 to 2012 had an economic impact of more than $70 million [51]. Others predict worse: for every hour it is not up and running, a cloud service can take a hit between $1 to 5 million [90].

Unfortunately, proving cloud services' dependability is challenging. Behind the cloud computing, it is backed by large sophisticated distributed software stack [35, 38, 39, 43, 44, 45, 58, 73, 84, 100, 131] that is running on top of large-scale cluster. Such cloud distributed systems remain difficult to get right because they need to address data races among machines, complex failures that randomly happen, tremendous user requests, and much more issues that caused from cloud computing infrastructure.

Data races are known to be a core problem in any concurrent software systems. Unlike non-distributed software, cloud distributed systems are subject to not only local concurrency bugs, which basically come from thread interleaving, but also distributed concurrency bugs, which come from inter-node message interleaving. Moreover, cloud hardware is built from commodity hard-

ware that failures can happen at anytime and can be very complex. The timing of these hardware failures plus message interleaving makes it hard to handle the concurrency correctly.

Moreover, the size of cloud users is tremendous and cloud service providers need to guarantee service quality (*i.e.*, availability and performance) to their users. The providers need to ensure that their capabilities can satisfy the current users and also make sure there is no glitch when users are growing. Cloud providers normally employ large-scale systems to achieve high aggregate capabilities, but large-scale systems are challenging to build and costly to test their correctness.

Addressing these challenges makes the systems getting more complex. New intricate bugs continue to happen and create dependability problems. Guaranteeing dependability has proven to be challenging in these systems [61, 63, 121, 126]. This raises a vital question: "*how can we make cloud-scale distributed systems reaching ideal dependability?*" We try to answer this question by focusing on the problems of distributed concurrency bugs and scalability bugs. These two are critical problems because they are novel issues that occur in cloud environment only and not many works addressing them. The following sections discuss our contributions to address the challenges.

1.1 Distributed Concurrency Bugs

Distributed concurrency bugs (DC bugs) are bugs that caused by nondeterministic orders of distributed events. Distributed events could be message arrivals, hardware crashes/reboots, network timeout, etc. Cloud systems execute multiple complicated distributed protocols concurrently (*e.g.*, serving users' requests, operating background tasks, and combined with untimely hardware failures), and possible interleavings of the distributed events are beyond developers' anticipation, which some interleavings might not be handled properly, and can cause catastrophic failures such as data loss/inconsistency and downtimes. Compared to the "countless" of efforts in combating "local" concurrency bugs in multi-threaded software, DC bugs have not received the same amount of attention within the research community.

Here is our contributions to combat DC bugs in systematic and comprehensive manners:

1. Bug study and taxonomy for DC bugs (TaxDC): we perform an in-depth study of more than 100 real-world DC bugs and built a first complet taxonomy for DC bugs. This study will give insight to guide many future research work on DC bugs.

2. Semantic-aware model checking (SAMC): we advance the state of the art of model checking for distributed systems by adopting white-box approach to tackle state-space explosion, the current limitation of model checking.

The brief detail of these two works are discussed below.

1.1.1 DC Bug Study & Taxonomy

Bug and failure studies can significantly guide many aspects of dependability research. Many researchers have recently employed formal studies on bugs and failures [75, 86, 87, 95, 110, 117, 124, 127]. However, we are not aware of any public large-scale DC-bug study, a recent study from Microsoft analyzed the effect of distributed concurrency of workload and only studied five DC bugs in MapReduce [124], and researchers from NEC Labs dissected only network-failure-related DC bugs to study and did not publicly release it [78].

In this dissertation, we fill the void by performing large-scale DC-bug study. We study 104 real-world DC bugs from four various popular cloud-scale distributed systems: Cassandra, HBase, Hadoop MapReduce/Yarn, and ZooKeeper. We study DC bugs in all aspects including trigger, errors and failures, and fixes.

For triggering conditions, we study DC bugs from two perspectives:

1. Timing conditions: 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 local concurrency bugs.

2. Input preconditions: 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 local concurrency 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.

Other than the trigger, we also look into errors and failures. 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. We categorize first errors into *local* errors and *global* errors, based on whether they can be observed from the triggering node alone. And after the first errors, we track down to system failures that are noticeable to users such as downtimes, lost/corrupted/inconsistent data, failed operations, and degraded performance. Identifying errors and failures help failure diagnosis get closer to disclosing bug triggering and root causes and help bug detection get closer to accurately predict failures.

Lastly, we study how developers fix DC bugs to understand their fix strategies. We want to see how different DC bug fixes compared to local concurrency bugs. 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. Understanding the fix strategies will help research on runtime failure prevention and automatic bug fixing.

Our contribution from the study is the first complete taxonomy of DC bugs which named TaxDC. TaxDC contains in-depth characteristics of DC bugs, stored in the form of 2,083 classification labels and 4,528 lines of re-enumerated steps to the bugs that we manually added. And as mentioned above, TaxDC can guide various future research on combating DC bugs such as model checking, bug detections, failure diagnosis, and failure prevention and fixing.

1.1.2 Semantic-Aware Model Checking

One powerful method for discovering hidden DC bugs is the use of an *implementation-level distributed system model checker* (**dmck**). A dmck can discover buggy interleavings that lead to DC bugs by reordering every possibility of nondeterministic distributed events. The last ten years have seen a rise of dmcks such as MaceMC, MODIST, or Demeter. One big challenge faced by a dmck is the state-space explosion problem (*i.e.*, there are too many distributed events to re-order). To address this, existing dmcks adopt a random walk or basic reduction techniques such as dynamic partial order reduction (DPOR). Despite these early successes, existing approaches cannot unearth many real-world DC bugs, so we advance state of the art of dmck to combat DC bugs.

We start by addressing two limitations of existing dmcks. First, existing dmcks treat every target system as a complete *black box*, and perform unnecessary reorderings of distributed events that would lead to the same states (*i.e.*, redundant executions). Second, they do not incorporate complex multiple fault events (*e.g.*, crashes, reboots, etc.) into their exploration strategies, as such inclusion would exacerbate the state-space explosion problem.

To address these limitations, we introduce Semantic-Aware Model Checking (**SAMC**), a novel white-box model checking approach that takes *semantic knowledge* of how distributed events (specifically, messages, crashes, and reboots) are processed by the target system and incorporates that to create reduction policies. The policies are based on sound reduction techniques such as DPOR and symmetry. The policies tell SAMC not to re-order some pairs of events such as message-message pairs, and message-crash pairs, yet preserves soundness, because those cut out re-orderings are redundant, and unnecessary to check.

SAMC can reproduce twelve old bugs in three cloud distributed systems (Cassandra, Hadoop MapReduce, and ZooKeeper) involving 30-120 distributed events and multiple crashes and reboots. Some of these bugs cannot be unearthed by non-SAMC approaches, even after two days. SAMC can find the bugs up to 340 (49x on average) faster compared to state-of-the-art techniques, it found two new bugs in Hadoop MapReduce and ZooKeeper.

1.2 Scalability Bugs

Scalability bugs are a type of bug that newly born in the era of cloud computing. These bugs are latent such that they do not surface in small/medium-scale deployments, but only surface in large scale. They threaten systems reliability and availability at scale. As we discussed above, cloud backend needs to be scalable; algorithms and protocols in cloud distributed systems are designed to be scalable. However, until real deployment takes place, if developers do not have a large cluster to test their actual implementations, unexpected bugs are unforeseen.

The following is our contribution to tackle this novel type of bugs:

1. Scalability bug study (SCB): we perform an in-depth study of 41 scalability bugs to analyze how an era of cloud computing gives a birth to a new type of bugs that is scale dependent. This study is a bug benchmark for future research on scalability aspect of cloud-scale distributed systems.

2. Scalability checking methodology for cloud-scale distributed systems (SCK): we propose a methodology to help developers test and debug scalability of systems in an economical way by colocate multiple nodes on one machine.

The next sections discuss some detail of these two works.

1.2.1 Scalablity Bug Study

For scalability bugs, the situation is worse. We are not aware of any study on scalability bugs at all, so in this dissertation, we start a study of scalability bugs to gain some foundational knowledge about them. We studied 41 bugs in seven systems including Cassandra, Couchbase, Hadoop MapReduce, HBase, HDFS, Riak, and Voldemort.

From the study, here is our observations regard to scalability bugs:

- Scalability bugs only appear at extreme scale.

- Systems can be scalable in design, but not in practice.
- Scalability bugs could be implementation specific and hard to predict.
- Scalability bugs are caused from cascading impacts of "not independent" nodes.
- It is long and difficult to debug large-scale.
- Not all developers have large test budgets.

1.3 Summary of Contributions and Outline

We summarize our contributions and present the outline for the rest of dissertation below.

- **Background:** Chapter 2 discusses a background of cloud computing and cloud-scale distributed systems, and how concurrency and scalability affect system developing.
- **Distributed concurrency bug study and taxonomy:**: Chapter 3 introduces TaxDC, a DC-bug study and taxonomy. The study provides insights how DC bugs are triggered, how they affect systems, and how they are fixed. These insights will help future research on combating DC bugs including model checking, bug detection, formal verification, failure diagnosis, and failure prevention.
- **Semantic-aware model checking:** Chapter 4 presents SAMC, a white-box approach to model check distributed systems. The approach advances the state-of-the-art of leveraging semantic knowledges to prune out redundant executions to tackle state space explosion. SAMC introduces some principles of semantic awareness that can mitigate the problem and this chapter also shows SAMPRO, a prototype of how SAMC approach can be adopted to Cassandra, Hadoop, and ZooKeeper.
- **Scalability bug study:** Chapter 5 presents SCB, a study of scalability bugs in cloud-scale distributed systems. The study provisions foundational knowledge of how a new type of

bugs happen when systems are targeted to deploy on large scale (*e.g.*, 500 nodes). This is the first formal study on scalability bugs.

- **Scalability checking methodology:** Chapter 6 presents SCK, a methodology to check scalability of systems in an economical manners. SCK introduces an approach for developers to colocate multiple CPU-intensive nodes on one machine to test and debug scalability bugs. With SCK, developers can mitigate a resource contention problem and yield the similar test and debug outcome as deploying systems on real large scale.

- **Conclusion and future work:** Chapter 7 concludes this dissertation, summarizes lessons learned, and discuss research avenues to make research work in this dissertation more complete.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter provides the background of this dissertation. We briefly discuss about cloud computing and cloud backend in Section 2.1. Section 2.2 and 2.3 discuss about distributed concurrency and scalability respectively. These are two aspects of distributed systems that could threaten system dependability. And in those two sections (Section 2.2 and Section 2.3), we also briefly discuss how system community address issues from distributed concurrency and scalability.

2.1 Cloud Computing and Cloud Backend

2.1.1 Cloud Computing

In the past decase, "cloud computing" has become widespread buzzword that IT people often talk about. However, there is few agreement on what it really means; many companies and experts have given many definitions [28, 29, 30, 55] that are different in details, but they share some common characteristics: cloud computing is on-demand internet-based services that can scale to serve growing users' requests, and users "pay as they go".

Cloud computing can attract a number of users to move their data and computation from local machines and private datacenters to the cloud [2, 8, 17, 19, 24, 26, 27, 98]. It provides many benefits as shown below:

- Users can access their computing resources and data any time and any where. For example, Google Doc allows users to create/modify their documents on one machine and access them later on mobile devices. This advantage also enables new classes of applications, especially, mobile applications [15, 16, 18, 22, 25]

- Users can cut the cost of hardware investment, but can get immedate access to computing resources. Moreover, this will help improve hardware utilization. Users do not need to pay

while they are not using the resources [99].

- Most importantly, cloud computing help users to scale their services in a convenient manner. When their workload is growing up, they can just purchase more computing power from their cloud service providers.

2.1.2 Cloud Backend

Behind the cloud computing, it is sophisticated distributed systems running on large-scale clusters. Unlike traditional computing, when the number of users or workload increase, we do not upgrade hardware specification, but add more machines to the systems. Thus, cloud distributed systems are designed to be scalable to handle tremendous users' requests. Moreover, the cluster is built from commodity machines which hardware failures are not optional and can be very complex (*e.g.*multiple machine failures, disk failures, and network partitions), so cloud systems must be able to handle these complex failures. We will discuss some types of cloud systems below as these systems are systems that we work on in this dissertation.

- *Distributed file systems*: these are file systems that stores files across machines in clusters (however, some systems cannot be mounted as other traditional file systems, and some users consider them as data store [9]). Distributed file systems do replication or erasure coding across machines in order to be fault tolerant, and to increase performance via data aggregation. Exmaples of these systems are GFS from Google, and HDFS from Apache which is open-source version of GFS.

- *Data-parallel framework*: these frameworks process "big data" by leverage parallelism. It eases parallel computing by enabling users to increase computing power by just adding more machines without changing their programs. One of well-known data-parallel frameworks is MapReduce from Google. As its name states, MapReduce is a programming model that consist of *map* and *reduce* functions. Map functions process key-value pairs of data and

generate intermediate key-value pairs, which reduce functions will values of data with the same keys. Apache also has open-source framework, which is similar to Google MapReduce, called Hadoop MapReduce.

- *NoSQL data stores*: These are data stores that are not relational database. Storing and accessing data do not have strict tabular relations like in relational database, and are not done by SQL query (some systems support SQL query but not fully), such as key-value stores, document stores, and object stores. Most NoSQL stores adopt the concept of "eventual consistency" to improve availablity during network partition (favoring "C" in CAP theorem). An example of NoSQL is Dynamo from Amazon. Dynamo is decentralized distributed key-value store. Its open-source counterpart from Apache is Cassandra.

- *Synchronization services*: These are utility services that support other large distributed systems. They help nodes in other systems synchronize some metadata such as global locking, configuration maintaining, and naming. Examples of the synchronization services are Chubby from Google and ZooKeeper from Apache.

2.2 Distributed Concurrency

2.2.1 Local Concurrency and Distributed Concurrency

A well-known concurrency that developers most familiar with is from thread execution interleaving in multi-threaded software or what we call in this dissertation "local concurrency". Multi-threaded software has become common in the age of multi-core processor, however, building multi-threaded software is hard. Developers need to handle all possible interleaving of multiple threads that are accessing to same data properly, otherwise concurrency bugs will happen. These bugs are timing-related and non-deterministic, and they are extremely difficult to test and debug.

For distributed systems, other than local concurrency, the systems are also subject to "distributed concurrency" that is caused from interleaving of computations in multiple nodes. Nodes

ZooKeeper Bug #1264:

1. *Follower F crashed* in the past,
2. *F reboots* and joins the cluster; then *F synchronizes data* with Leader L
3. F sends FOLLOWERINFO message to L [synchronization message]
4. L sends LEADERINFO message to F [synchronization message]
5. F sends ACKEPOCH message to L [synchronization message]
6. L sends SNAP message to F [synchronization message]
7. L sends data tree snapshot to F [synchronization message]
8. L sends NEWLEADER message to F [synchronization message]
9. *Client C sends a request* to update data with Tx-#15 to L; L does atomic broadcast to update all followers
10. L sends update proposal message for Tx-#15 to F [broadcast message]
11. F sends update ack message for Tx-#15 to L [broadcast message]
12. *L sends update commit message* for Tx-#15 to F [broadcast message]
13. *F applies the update* for Tx-#15 to in-memory data tree, but not to on-disk log (because F has not received UPTODATE message)
14. *L sends UPTODATE message* to F [synchronization message]
15. C sends a request to update data with Tx-#16 to L
16. L sends update proposal for Tx-#16 to F
17. F sends update ack for Tx-#16 to L
18. L sends update commit for Tx-#16 to F
19. F applies the update for Tx-#16 to in-memory data tree and on-disk log
20. *F crashes* (before *F does snapshot*)
21. F reboots and joins the cluster again
22. L synchronized data with F by sending update starting from Tx-#17
23. F loses the update for Tx-15 C did in step 9

Figure 2.1: **ZooKeeper-1264 bug.** *A concurrency bug in ZooKeeper that is caused from a mix of untimely message arrivals and crash timing. This bug surfaces when a follower receives update commit messasge (step 12) in the middle of an atomic operation (step 3-14) and the follower crashes before it does snapshot (step 20)*

in distributed systems do not have shared memory and they access data in other nodes via network communication, so basically, distributed concurrency comes from concurrent message arrivals and internal computations in running nodes.

Other than timing of message arrivals and local computations, concurrent bugs in distributed systems are caused from timing of failures as well. Cloud software is often deploy on commodity hardware for vertical scaling purpose (Section 2.3.1). This commodity hardware is unreliable, and hardware failures are not an option. Cloud distributed systems need to response to these failures, they need to detect and recover from the failures and makes sure that users's data will not be lost or corrupted. Gauranteeing this correctness is proven to be hard, cloud-scale distributed systems need to handle failures that can happen at any time and at any state of the systems. Some ordering of message arrivals could make systems into state that developers never anticipate and is prone to error when failure happens.

Consider hardware failures, concurrency bugs in cloud-scale distributed systems is not only about interleaving of message arrivals and local computations, but also timing of hardware failures as well. Figure 2.1 shows an example of a concurrency bug that happens because of untimely ordering of message arrivals and node crashes. This bugs surfaces only if a follower receives an UPTODATE message (step 14) after a commit message (step 12), and the follower crashes before it does snapshot (step 20); only untimely message arrivals or the timing of follower crash is not enough for bug to surface.

2.2.2 Distributed Systems Model Checker (DMCK)

In order to unearth DC bugs the question we have to answer is: "*can we exercise necessary conditions (i.e.workloads and faults) and test different event re-ordering to hit the bugs?*". This is the job of distributed system model checkers (dmck), which are gaining popularity recently [63, 82, 114, 126]. Dmck works by intercepting distributed events and permuting their ordering, and hereby pushing the target system into corner-case situations and unearthing hard-to-find bugs.

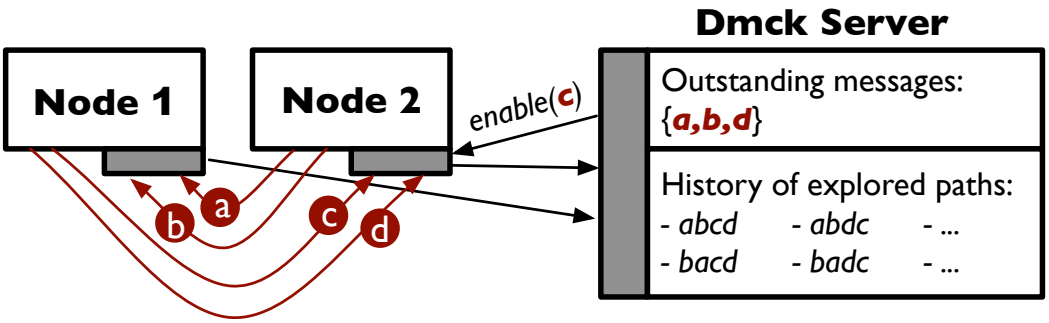

Figure 2.2: **DMCK.** *A typical framework of a distributed system model checker (dmck).*

However, the more events included, the more scalability issues will arise due to state-space explosion.

The last ten years have seen a rise of software model checker that checks distributed systems directly at the implementation level. Figure 4.1 illustrates a dmck integration to a target distributed system, a simple representation of existing dmck frameworks [63, 82, 114, 126]. The dmck inserts an interposition layer in each node of the target system with the purpose of controlling all important events (*e.g.*, network messages, timeouts) and preventing the target system to process the events until the dmck enables them. A main dmck mechanism is the permutation of events; the goal is to push the target system into all possible ordering scenarios. For example, the dmck can enforce `abcd` ordering in one execution, `bcad` in another, and so on.

2.3 Scalability

2.3.1 Vertical Scaling vs Horizontal Scaling

When systems' workload grows (the number of users raises or individual users' requests increase), developers need to scale the systems to add more capability and keep users satisfied. Two traditional approaches to scale system are used as we show below [101]:

- **Vertical scaling or scale-up**: this approach expands system capabilities by adding more resources (*e.g.*, CPU, memory, and storage) to a single node to boost its performance, and make

software to leverage additional resources. For example, run more processes of applications in the node.

- **Horizontal scaling or scale-out**: this approach enhance the capability by adding more nodes to current distributed systems to yeild higher aggregate capability; mostly, the nodes that we are adding are low-cost machines.

In the past, vertical scaling was widely favored by many companies. Multiprocessor with higher clock rate can satisfy computing power need of largest companies [101]. Vertical scaling requires less human effort than horizontal scaling; it does not need more administrative effort because the number of machines and systems administrators need to handle is still the same. The disadvantages of vertical scaling are the upgradability is limited by existing hardware manufacturing, and the upgrade cost is expensive.

Because of the upgradability and price issues, nowadays, the trend goes to horizontal scaling. Many cloud service companies adopt this approache (*e.g.*, Google, Facebook, Amazon, etc.). The cost of horizontal scaling is much more cheaper than the vertical scaling and there is not limitation for hardware to scale out infinitely (the limitations are posed by software stack) [14]. In addition, hardware manufaturers try to facilitate scale-out approach [101].

2.3.2 Scalability Testing

As we discussed in 2.3.1, horizontal scaling or scale-out is a trend now because it does not expose hardware limitation. The limitation is in software stack so developers need to invent scalable algorithms and protocols, however, before real deployment, if they do not have a large cluster to test their implementations, there could be "scalability bugs" hide there.

We now discuss popular approaches (simulation, extrapolation, and emulation) for unearthing scalability bugs that avoid acquiring a number of machines, because testing on such deployments is costly. First, simulation approaches test system/application models in different scales [37, 83], However, a model can look scalable but the actual implementation can contain unforeseen bugs.

Later in Section 5, we will show our observations from *real world* scability bugs that simulation cannot detect them.

Second, extrapolation monitors system behaviors in "mini clusters" and extrapolates them to larger scales (Section 2.1 in [121]). However, mini clusters tend to be order(s) of magnitude smaller than real deployments. Most importantly, system behaviors do not always extrapolate linearly [121].

Finally, real-scale emulation checks real implementations in an emulated environment [64, 121]. This approach emulates real large-scale system in a single machine. For example, a naive way to achieve this is just colocating multiple processes/VMs on one machine. The limitation here is emulation consumes real resources (CPU, memory, and storage), so with limited resources, we can not emulate *really large* deployment (*e.g.*, can test up to 50-node deployment). Some works try to resolve this resource contention issue [64, 121]. We will discuss them in related work (Section **??**).

2.4 Conclusion

In this chapter, we discuss about cloud-scale distributed systems, software backend for the cloud computing. We show what the current trend of the systems is and how they are design. We also discuss about distributed concurrency and scalability, the two important aspects of cloud-scale distributed systems that could threaten dependability of the systems. We briefly discuss how system community address issues from the concurrency and scalability. Unfortunately, distributed concurrency bugs are still an unsolved problem, and scalability bugs are novel and not many works address about them.

CHAPTER 3

TAXDC: A TAXONOMY OF NON-DETERMINISTIC CONCURRENCY BUGS IN CLOUD DISTRIBUTED SYSTEMS

Concurrency bugs are one notorious type of software bugs that happen in concurrency systems. 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. And for cloud-scale distributed systems, the reliability is also 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 [47, 62]. 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, a comprehensive bug study is needed. Past studies have closely examined bugs in various software systems [40, 94, 104], which have motivated and guided many aspects of reliability research. There are few bug studies on cloud-scale distributed systems [62, 86], but they did not specifically dissect DC bugs. There was an internal bug study dissecting network-failure-related DC bugs to be a foundation to combat those bugs, but it was not published [78], and one recent work analyzed non-determinism in MapReduce programs but only discussed five bugs [124]. Thorough studies have also been conducted for LC bugs [54, 95] with many follow-up work to date, yet *there is no comprehensive study on real-world distributed concurrency bugs*.

We fill this void in this chapter by presenting out in-depth analysis of real-world DC bugs in

well-known cloud distributed systems, and introducing TaxDC, a comprehensive taxonomy of DC bugs that covers several axes. We briefly give an overview of TaxDC in Section 3.1, present our analysis in Section 3.3-3.7, and discuss some lessons we learn from the study in Section 3.8.

3.1 TaxDC

In this formal study on DC bugs, we do in-depth analysis of 104 DC bugs. The bugs came from four popular cloud distributed systems: Cassandra [3], HBase [6], Hadoop MapReduce [5], 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 3.1.

As the main contribution, TaxDC will be the first large-scale DC-bug benchmark. In the last six years, bug benchmarks for LC bugs have been released [74, 129], 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 postmortem debugging [56, 92, 93, 109], testing and model checking [63, 82, 85, 114, 126], and verifiable language frameworks [46, 123]. 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 3.8.

3.2 Methodology

3.2.1 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.

3.2.2 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) [5] 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 chapter, we will present short examples of DC bugs in these systems. Some detailed examples are illustrated in Figure 3.2, 3.4 and 3.5.

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 [1], 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.

3.2.3 Taxonomy

We study the characteristics of DC bugs along three key stages: triggering, errors & failures, and fixing (Table 3.1). *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.

3.2.4 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. Finally, We then *re-enumerate* the full sequence of operations (the "*steps*") to a clearer and more concise description such as the ones in Figure 3.2. 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

Triggering
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
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
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 3.1: **Taxonomy of DC Bugs.**

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 Section 3.3.1, 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.

3.2.5 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 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.

3.2.6 Detailed Terminologies

Below are the detailed terminologies we use in this chapter. 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 chapter, we present bug examples by abstracting system-specific names. As shown in Figure 3.1, 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,

1. http://ucare.cs.uchicago.edu/project/taxDC

we use "N*" to represent crash, "N!" reboot, and "N+" local computation at N.

We cite bug examples with clickable hyperlinks (*e.g.*, MR-3274). 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.

3.3 Trigger

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* (Section 3.3.1): 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* (Section 3.3.2): 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.

3.3.1 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 3.2 shows the per-system breakdown and Figure 3.3a (TC) the overall breakdown. Since a few bugs are triggered by more than one type of timing conditions (Section 3.3.3), the sum of numbers in Table 3.2 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 3.2) 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 3.2) 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 [95]; our study of DC bugs shows that this relationship does not apply or even gets reversed in several representative distributed systems.

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 3.1a illustrates ac/bc race at node C in MR-3274. 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 3.1b). Message-message race can also happen between a message arrival and a message sending. For example, the ab/bc race in Figure 3.1c depicts HB-5780. In this bug, B_{RS}

	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 3.2: **#DC bugs triggered by timing conditions (Section 3.3.1).** *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 3.3a).*

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 3.1d illustrates ab/ba race crisscrossing two nodes A and B in MR-5358. 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 3.1e).

An order violation can also originate from a race between a message and a local computation (*message-compute race*). For example, Figure 3.1f illustrates $ab/b+$ race in MR-4157. 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 3.1g).

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 3.1h illustrates MR-5009. 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

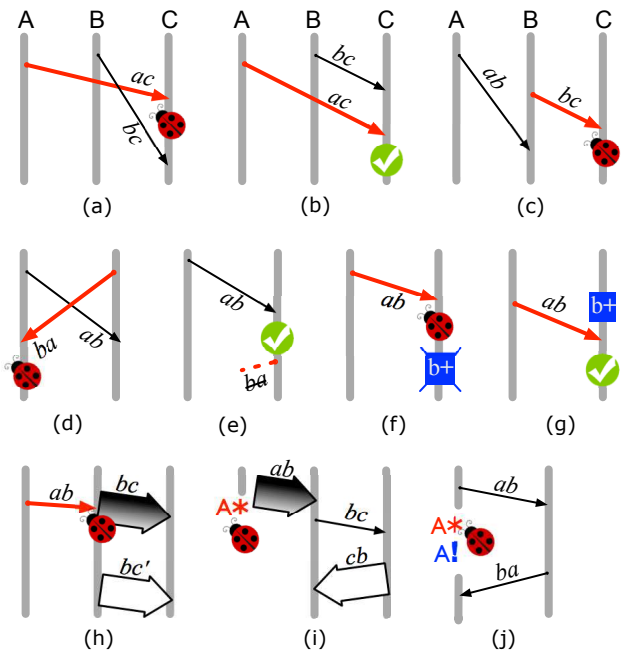

Figure 3.1: **Triggering patterns (Section 3.3.1).** *The three vertical lines represent the timeline of nodes A, B and C. An arrow with* `xy` *label implies a message from X to Y. A square box with label* `x+` *implies a local state-modifying computation at node X. A thick arrow implies a set of messages performing an atomic operation.* `X*` *and* `X!` *implies a crash and reboot at node X respectively (Section 3.2.6). All figures are discussed in Section 3.3.1*

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 3.2) 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 3.1i illustrates a fault-timing bug in MR-3858. 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 3.1j depicts a reboot-timing bug in MR-3186. 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.

Message-Fault Bugs. Four DC bugs are caused by a combination of messages and faults. For example, in Figure 3.2, a message (step 12) arrives in the middle of some atomic operation (step 3-14). This message atomicity violation leads to an error that further requires a fault timing (step 20) to become an externally visible failure.

> **Finding #1:** DC bugs are triggered mostly by *untimely messages* (64% in Table 3.2) 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.

3.3.2 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 3.2. First, a fault and a reboot (step 1-2) and a client request (step 9) must happen to create a path to the message atomicity violation (step 9 interfering with step 3-14). Second, conflicting messages from two different protocols (ZAB and NodeJoin initiated in step 2 and 9) have to follow specific bug-triggering timing conditions. Even after the atomicity violation (after step 14), the bug is not guaranteed to lead to any error yet (*i.e.*, a benign race). Finally, the follower experiences an untimely fault (step 20), such that after it re-

ZK-1264:

1. *Follower F crashed* in the past,
2. *F reboots* and joins the cluster; then *F synchronizes data* with Leader L
3. F sends FOLLOWERINFO message to L [synchronization message]
4. L sends LEADERINFO message to F [synchronization message]
5. F sends ACKEPOCH message to L [synchronization message]
6. L sends SNAP message to F [synchronization message]
7. L sends data tree snapshot to F [synchronization message]
8. L sends NEWLEADER message to F [synchronization message]
9. *Client C sends a request* to update data with Tx-#15 to L; L does atomic broadcast to update all followers
10. L sends update proposal message for Tx-#15 to F [broadcast message]
11. F sends update ack message for Tx-#15 to L [broadcast message]
12. *L sends update commit message* for Tx-#15 to F [broadcast message]
13. *F applies the update* for Tx-#15 to in-memory data tree, but not to on-disk log (because F has not received UPTODATE message)
14. *L sends UPTODATE message* to F [synchronization message]
15. C sends a request to update data with Tx-#16 to L
16. L sends update proposal for Tx-#16 to F
17. F sends update ack for Tx-#16 to L
18. L sends update commit for Tx-#16 to F
19. F applies the update for Tx-#16 to in-memory data tree and on-disk log
20. *F crashes* (before *F does snapshot*)
21. F reboots and joins the cluster again
22. L synchronized data with F by sending update starting from Tx-#17
23. F loses the update for Tx-15 C did in step 9

Figure 3.2: **A DC bug in ZooKeeper.** *This figure shows Figure 2.1 again. It shows a DC bug in ZooKeeper that is caused from a mix of untimely message arrivals and crash timing. This bug surfaces when a follower receives update commit messasge (step 12) in the middle of an atomic operation (step 3-14) and the follower crashes before it does snapshot (step 20)*

boots (step 21), a global replica-inconsistency error will happen (step 23). Put it in a reverse way, before step 20, 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 3.3b (FLT) shows that 63% of DC bugs must have at least one fault. In more detail, Figure 3.3c-e (TO, CR, 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 3.3f (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 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 3.3g (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 3.3b), multiple protocols (80% in Figure 3.3f), and background protocols (81% in Figure 3.3g) .

3.3.3 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),

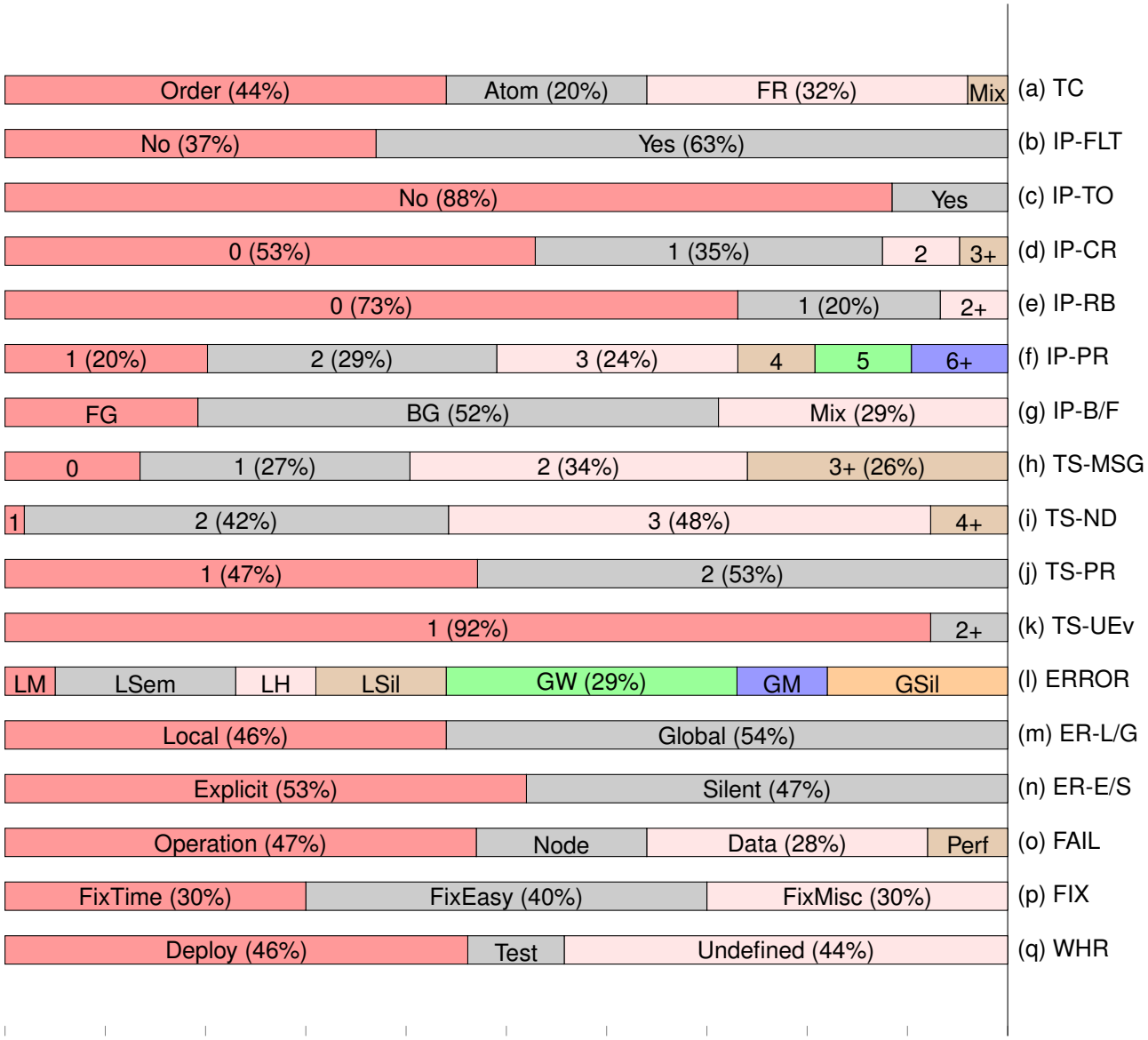

Figure 3.3: **Statistical overview of TaxDC.** *Timing Conditions (TC) is discussed in Section 3.3.1, Input Preconditions (IP) in Section 3.3.2, Triggering Scope (TS) in Section 3.3.3, Errors (ER) in Section 3.4.1, Failures (FAIL) in Section 3.4.2, Fixes (FIX) in Section 3.5, and Where Found (WHR) in Section 3.7.*

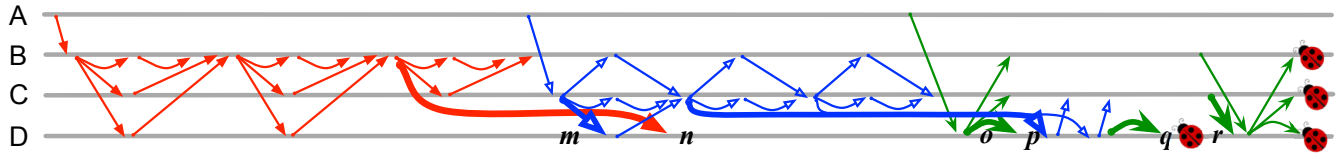

Figure 3.4: **A Cassandra's Paxos bug.** *In CA-6023, three key-value updates (different arrow types) concurrently execute the Paxos protocol on four nodes (we simplify from the actual six nodes). The bug requires three message-message race conditions: (1)* m *arrives before* n*, (2)* o *before* p*, 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.*

protocol (TS-PR), and untimely event (TS-UEv) counts as shown in Figure 3.3h-k. This statistic is important with respect to the scalability of model checking, bug detection and failure diagnostic tools (Section 3.8.2-3.8.5).

Message count implies the minimum number of messages involved in E as defined in the beginning of section 3.3. Figure 3.3h (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 3.1a, 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 3.3i-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 3.3k (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 9 and 20 in Figure 3.2). Three are fault-reboot timing bugs, each requiring one untimely fault and one untimely reboot. The last one is CA-6023, shown in Figure 3.4, 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 3.3h-j). Most DC bugs are mostly triggered by only *one* untimely event (92% in Figure 3.3k).

3.4 Errors and Failures

3.4.1 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 (Section 3.8).

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 3.1a) or the node with untimely fault (*e.g.*, node A in Figure 3.1i). For each error, we also check whether it is an *explicit* or *silent* error. Table 3.3 and Figure 3.3l (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 3.3) 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 mes-*

	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 3.3: **First error symptoms of DC bugs (Section 3.4.1).** *Some bugs cause multiple concurrent first errors.*

sages (29% in Table 3.3). 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 3.3m (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 3.3n (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%).

3.4.2 Failure Symptoms

Figure 3.3o (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.

3.5 Fixes

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.

3.5.1 Message Timing Bug Fixes

The left half of Table 3.4 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 3.1f, MR-5465'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, HB-5780's fix for ab/bc race in Figure 3.1c forces the sending of bc request at B to wait for the receipt of ab; the result is ab-bc local serialization at B.

The right half of Table 3.4 shows that fix handling is more popular. Fix handling fortunately

	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 3.4: **Fix strategies for message timing bugs (Section 3.5.1).** *Some bugs require more than one fix strategy.*

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 3.1a, MR-3274 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 3.1d, MR-5358 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, MR-2995'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. MR-5198'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.

3.5.2 *Fault/Reboot Timing Bug Fixes*

Table 3.5 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 (HB-3596). Here, as A shuts down earlier, B assumes responsibility of A's regions (via a

	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 3.5: **Fix strategies for fault/reboot timing bugs (Section 3.5.2).** *Some bugs require more than one fix strategy.*

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 3.5). 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 3.3p), 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%).

3.6 Root Causes

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 3.1a. 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 3.1f. 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.

3.7 Other Statistics

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 Section 3.2.4, (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 3.3r (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.

3.8 Lessons Learned

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.

3.8.1 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 [61, 79, 130], we emphasize that all forms of faults (Section 3.2.6) must be exercised.

3.8.2 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 (Section 3.3.1)? This is the job of distributed system model checkers (dmck), which are gaining popularity recently [63, 82, 85, 114, 126]. 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 [82] only reorders messages and network disconnections. MoDist [63] exercises timeouts and Demeter [63] intercepts messages and local computation but they do not explore different timing of multiple crashes and reboots. SAMC [85] exercises multiple faults but does not support timeout and thread controls. Also, none of the above include storage faults or timing issues [66]. 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 [102].

3.8.3 *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 [63] and SAMC [85] 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 [61, 92, 109]. Developers also bemoan the hard-to-debug fail-silent problems MR-3634 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 Section 3.8.4). 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 3.4). 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.

3.8.4 *Bug Detection Tools*

We now discuss bug detection tools, which are unfortunately rare for DC bugs, although very popular for LC bugs [33, 52, 71, 96, 102, 111]. 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 [49, 50, 96]. 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 [49]; "the atomicity of accesses a_1 and a_2 should never be violated" is effective for LC bugs [96]. 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"* [96] and *"a condition checked to be true should remain true when used"* [107]. 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 [34, 105, 112], there have been few for DC bugs [113]. 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 3.3d-f) can be found with inputs of at most two protocols, crashes and reboots.

3.8.5 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 [88] and concurrency bugs [76] 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 3.1b happens 99.99% of the time while Figure 3.1a 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 [41]. 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 [106, 119]. 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 (ZK-2172). 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 [57, 93].

3.8.6 *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 [81, 97, 132]. 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 3.1a–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 [77, 91, 122]. 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 computation, without adding new computation logic. This presents a unique opportunity for developing new and more aggressive fixing techniques.

3.8.7 *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,

HB-9095:

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 3.5: **Race of HBase's messages to ZooKeeper.**

the *actual implementations of theoretically-proven distributed transactions are not always correct* (as also alluded in other work [35**?**]). 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 3.5; 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 [72]).

3.8.8 *Verifiable Frameworks*

Recently there is a growing work on new programming language frameworks for building verifiable distributed systems [46, 67, 123], 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.

3.8.9 *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 [108, 71] 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**).

3.9 Conclusion

Here is our summarized 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 run-time 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.

CHAPTER 4

SAMC: SEMANTIC-AWARE MODEL CHECKING FOR FAST DISCOVERY OF DC BUGS IN CLOUD DISTRIBUTED SYSTEMS

As we discuss in Section 2.2.2, a recent popular approach to unearth DC bugs is adopting distributed system model cheker or dmck. However, due to the complexity of DC bugs that we discuss in Chapter 3 (*e.g.*, interactions of multiple protocols, and various and multiple faults), the state of the art of dmcks still cannot work effectively [63, 82, 114, 126]. We observe that the existing systematic reduction policies cannot find bugs quickly, and cannot scale with the inclusion of fault events.

In this chapter, we discuss how to advance the state of the art by leveraging semantic awareness to assist model checking, and introduce "Semantic-Aware Model Checking" (SAMC), a white-box model checking approach that takes semantic knowledge of how events (*e.g.*, messages, crashes, and reboots) are processed by the target system and incorporates that information in reduction policies. We first discuss background of dmck and the state of the art in Section 4.1, then we discuss about the concept of SAMC and its implementation in Section 4.2 and 4.3 respectively. We evalute SAMC by comparing with the state of the art in Section 6.3, and lastly, we discuss other issues in adopting SAMC to check distirbuted systems in Section 4.5.

4.1 Background

This section gives more background on dmck and related terms, followed with a detailed overview of the state of the art.

4.1.1 DMCK Framework and Terms

As mentioned before, we define *dmck* as software model checker that checks distributed systems directly at the implementation level. Figure 4.1 illustrates a dmck integration to a target distributed

system, a simple representation of existing dmck frameworks [63, 82, 114, 126]. The dmck inserts an interposition layer in each node of the target system with the purpose of controlling all important events (*e.g.*, network messages, timeouts) and preventing the target system to process the events until the dmck enables them. A main dmck mechanism is the permutation of events; the goal is to push the target system into all possible ordering scenarios. For example, the dmck can enforce `abcd` ordering in one execution, `bcad` in another, and so on.

We now provide an overview of basic dmck terms we use in this paper and Figure 4.1. Each node of the target system has a *local state* (`ls`), containing many variables. An *abstract local state* (`als`) is a subset of the local state; dmck decides which `als` is important to check. The collection of all (and abstract) local states is the *global state* (`gs`) and the *abstract global state* (`ags`) respectively. The *network state* describes all the *outstanding messages* currently intercepted by dmck (*e.g.*, `abd`). To model check a specific protocol, dmck starts a *workload driver* (which restarts the whole system, runs specific workloads, etc.). Then, dmck generates many (typically hundreds/thousands) executions; an *execution* (or a *path*) is a specific ordering of events that dmck enables (*e.g.*, `abcd`, `dbca`) from an initial state to a termination point. A *sub-path* is a subset of a path/execution. An *event* is an action by the target system that is intercepted by dmck (*e.g.*, a network message) or an action that dmck can inject (*e.g.*, a crash/reboot). Dmck enables one event at a time (*e.g.*, `enable(c)`). To permute events, dmck runs *exploration methods* such as brute-force (*e.g.*, depth first search) or random. As events are permuted, the target system enters hard-to-reach states. Dmck continuously runs state *checks* (*e.g.*, safety checks) to verify the system's correctness. To reduce the state-space explosion problem, dmck can employ *reduction policies* (*e.g.*, DPOR or symmetry). A policy is *systematic* if it does not use randomness or bug-specific knowledge. In this chapter, we focus on advancing systematic reduction policies.

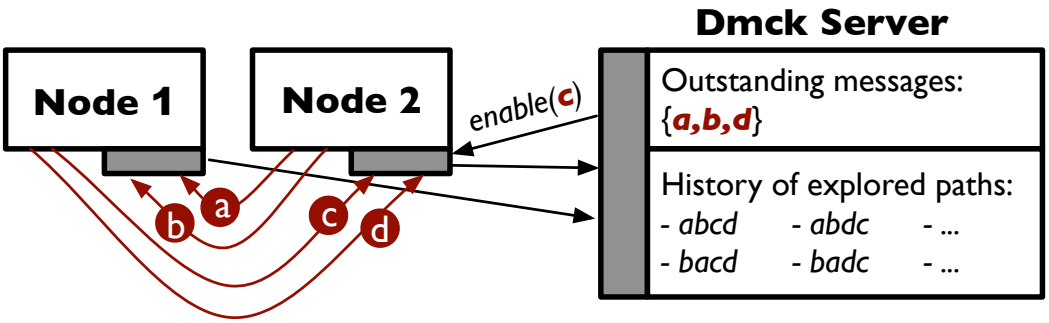

Figure 4.1: **DMCK.** *A typical framework of a distributed system model checker (dmck).*

4.1.2 State-of-the-Art DMCKs

MODIST [126] is arguably one of the most powerful dmcks that comes with systematic reduction policies. MODIST has been integrated to real systems due to its exploration scalability. At the heart of MODIST is *dynamic partial order reduction (DPOR)* [53] which exploits the *independence* of events to reduce the state explosion. Independent events mean that it does not matter in what order the system execute the events, as their different orderings are considered equivalent.

To illustrate how MODIST adopts DPOR, let's use the example in Figure 4.1, which shows four concurrent outstanding messages `abcd` (`a` and `b` for `N1`, `c` and `d` for `N2`). A brute-force approach will try all possible combinations (`abcd`, `abdc`, `acbd`, `acdb`, `cabd`, and so on), for a total of 4! executions. Fortunately, the notion of event independence can be mapped to distributed system properties. For example, MODIST specifies this reduction policy: a message to be processed by a given node is independent of other concurrent messages destined to other nodes (based on vector clocks). Applying this policy to the example in Figure 4.1 implies that `a` and `b` are dependent[1] but they are independent of `c` and `d` (and vice versa). Since only dependent events need to be reordered, this reduction policy leads to only 4 executions (`ab-cd`, `ab-dc`, `ba-cd`, `ba-dc`), giving a 6x speed-up (4!/4).

Although MODIST's speed-up is significant, we find that one scalability limitation of its DPOR application is within its *black-box* approach; it only exploits general properties of distributed sys-

1. In model checking, "dependent" events mean that they must be re-ordered. "Dependent" does not mean "causally dependent".

tems to define message independence. It does not exploit any semantic information from the target system to define more independent events. We will discuss this issue later (Section **??**).

Dynamic interface reduction (DIR) [63] is the next advancement to MODIST. This work suggests that a complete dmck must re-order not only messages (global events) but also thread interleavings (local events). The reduction intuition behind DIR is that different thread interleavings often lead to the same global events (*e.g.*, a node sends the same messages regardless of how threads are interleaved in that node). DIR records local exploration and replays future incoming messages without the need for global exploration. In our work, SAMC focuses only on global exploration (message and failure re-orderings). We believe DIR is orthogonal to SAMC, similar to the way DIR is orthogonal to MODIST.

MODIST and DIR are examples of dmcks that employ advanced systematic reduction policies. LMC [60] is similar to DIR; it also decouples local and global exploration. dBug [114] applies DPOR similarly to MODIST. There are other dmcks such as MACEMC [82] and CrystalBall [125] that use basic exploration methods such as depth first (DFS), weight-based, and random searches.

Other than the aforementioned methods, *symmetry* is another foundational reduction policy [48, 116]. Symmetry-based methods exploit the architectural symmetry present in the target system. For example, in a ring of nodes, one can rotate the ring without affecting the behavior of the system. Symmetry is powerful, but we find no existing dmcks that adopt symmetry.

Besides dmcks, there exists sophisticated testing frameworks for distributed systems (*e.g.*, FATE [61], PREFAIL [79], SETSUDO [78], OpenStack fault-injector [80]). This set of work emphasizes the importance of multiple failures, but their major limitation is that they are not a dmck. That is, they cannot systematically control and permute non-deterministic choices such as message and failure reorderings.

4.1.3 *Does State of the-Art Help?*

We now combine our observations in the previous section and our insight from TaxDC (Chapter 3), and describe why state-of-the-art dmcks do not address present reliability challenges of cloud systems.

First, *existing systematic reduction policies often cannot find bugs quickly*. Experiences from previous dmck developments suggest that significant savings from sound reduction policies do not always imply high bug-finding effectiveness [63, 126]. To cover deep states and find bugs, many dmcks revert to non-systematic methods such as randomness or manual checkpoints. For example, MODIST combines DPOR with random walk to "jump" faster to a different area of the state space (Section 4.5 of [126]). DIR developers find new bugs by manually setting "interesting" checkpoints so that future state explorations happen from the checkpoints (Section 5.3 of [63]). In our work, although we use different target systems, we are able to reproduce the same experiences above (Section 4.4.1).

Second, *existing dmcks do not scale with the inclusion of failure events*. Given the first problem above, exercising multiple failures will just exacerbate the state-space explosion problem. Some frameworks that can explore multiple failures such as MACEMC [82] only do so in a random way; however, in our experience (Section 4.4.1), randomness many times cannot find deep bugs quickly. MODIST also enabled only one failure. In reality, multiple failures is a big reliability threat, and thus must be exercised.

We conclude that finding systematic (no random/checkpoint) policies that can find deep bugs is still an open dmck research problem. We believe without semantic knowledge of the target system, dmck hits a scalability wall (as also hinted by DIR authors; Section 8 of [63]). In addition, as crashes and reboots need to be exercised, we believe recovery semantics must be incorporated into reduction policies. All of these observations led us to SAMC, which we describe next.

4.2 SAMC

4.2.1 Semantic Awareness

In a simple leader election protocol, every node broadcasts its vote to reach a quorum and elect a leader. Each node begins by voting for itself (*e.g.*, `N2` broadcasts `vote=2`). Each node receives vote broadcasts from other peers and processes every vote with this simplified code segment below. As depicted in the code segment below, if an incoming vote is less than the node's current vote, it is simply discarded. If it is larger, the node changes its vote and broadcasts the new vote.

```
if (msg.vote < myVote) {
  discard;
} else {
  myVote = msg.vote; broadcast(myVote);
}
```

Let's assume `N4` with `vote=4` is receiving three concurrent messages with votes `1`, `2`, and `3` from its peers. Here, a dmck with a black-box DPOR approach must perform 6 (3!) orderings (`123`, `132`, and so on). This is because a black-box DPOR does *not* know the *message processing semantic* (*i.e.*, how messages will be processed by the receiving node). Thus, a black-box DPOR must treat all of them as dependent (Section 6.1); they must be re-ordered for soundness. However, by knowing the processing logic above, a dmck can soundly conclude that all orderings will lead to the same state; all messages will be discarded by `N4` and its local state will not change. Thus, a semantic-aware dmck can reduce the 6 redundant executions to just 1 execution.

The example above shows a scalability limitation of a black-box dmck. Fortunately, simple semantic knowledge has a great potential in removing redundant executions. Furthermore, semantic knowledge can be incorporated on top of sound model checking foundations such as DPOR and symmetry, as we describe next.

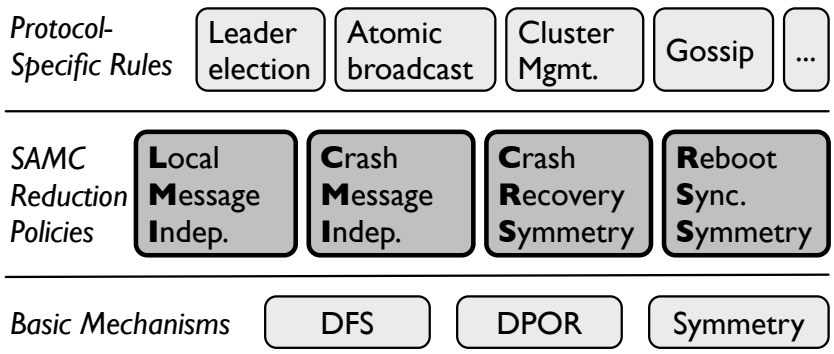

Figure 4.2: **SAMC Architecture.**

4.2.2 Architecture

Figure 4.2 depicts the three levels of SAMC: sound exploration mechanisms, reduction policies, and protocol-specific rules. SAMC is built on top of sound model checking foundations such as DPOR [53, 59] and symmetry [42, 116]. We name these foundations as mechanisms because a dmck must specify accordingly what events are dependent/independent and symmetrical, which in SAMC will be done by the reduction policies and protocol-specific rules.

Our main contribution lies within our four novel *semantic-aware reduction policies*: local-message independence (LMI), crash-message independence (CMI), crash recovery symmetry (CRS), and reboot synchronization symmetry (RSS). To the best of our knowledge, none of these approaches have been introduced in the literature. At the heart of these policies are *generic event processing patterns* (*i.e.*, patterns of how messages, crashes, and reboots are processed by distributed systems). Our policies and patterns are simple and powerful; they can be applied to many different distributed systems. Testers can extract the patterns from their target protocols (*e.g.*, leader election, atomic broadcast) and write protocol-specific rules in few lines of code.

In the next section, we first present our four reduction policies along with the processing patterns. Later, we will discuss ways to extract the patterns from target systems (Section 4.2.4) and then show the protocol-specific rules for our target systems (Section 4.3.2).

4.2.3 Semantic-Aware Reduction Policies

We now present four semantic-aware reduction policies that enable us to define fine-grained event dependency/independency and symmetry beyond what black-box approaches can do.

Local-Message Independence (LMI)

We define *local messages* as messages that are concurrently in flight to a given node (*i.e.*, intra-node messages). As shown in Figure 4.3a, a black-box DPOR treats the message processing semantic inside the node as a black box, and thus must declare the incoming messages as dependent, leading to 4! permutation of `abcd`. On the other hand, with white-box knowledge, local-message independence (LMI) can define *independency relationship among local messages*. For example, illustratively in Figure 4.3b, given the node's local state (`ls`) and the processing semantic (embedded in the `if` statement), LMI is able to define that `a` and `b` are dependent, `c` and `d` are dependent, but the two groups are independent, which then leads to only 4 re-orderings. This reduction illustration is similar to the one in Section 6.1, but this time LMI enables DPOR application on local messages.

LMI can be easily added to a dmck. A dmck server typically has a complete view of the local states (Section 4.1.1). What is needed is the *message processing semantic*: how will a node (`N`) process an incoming message (*m*) given the node's current local state (`ls`)? The answer lies in these four simple *message processing patterns* (discard, increment, constant, and modify):

```
Discard:                Increment:
if (pd(m,ls))           if (pi(m,ls))
 (noop);                  ls++;

Constant:               Modify:
if (pc(m,ls))           if (pm(m,ls))
  ls = Const;             ls = modify(m,ls);
```

In practice, `ls` and *m* contain many fields. For simplicity, we treat them as integers. The functions with prefix `p` are boolean read-only functions (predicates) that compare an incoming message (*m*) with respect to the local state (`ls`); for example, `pd` can return true if `m<s`. The first pattern is a *discard* pattern where the message is simply discarded if `pd` is true. This pattern is prevalent in distributed systems with votes/versions; old votes/versions tend to be discarded (*e.g.*, our example in Section **??**). The *increment* pattern performs an increment-by-one update if `pi` is true, which is also quite common in many protocols (*e.g.*, counting commit acknowledgements). The *constant* pattern changes the local state to a constant whenever `pc` is true. Finally, the *modify* pattern changes the local state whenever `pm` is true.

Based on these patterns, we can apply LMI in the following ways. (1) *x* is independent of *y* if `pd` is true on any of *x* and *y*. That is, if *x* (or *y*) will be discarded, then it does not need to be re-ordered with other messages. (2) *x* is independent of *y* if `pi` (or `pc`) is true on both *x* and *y*. That is, the re-orderings do not matter because the local state is monotonically increasing by one (or changed to the same constant). (3) *x* and *y* are dependent if `pm` is true on *x* and `pd` is not true on *y*. That is, since both messages modify the local state in unique ways, then the re-orderings can be "interesting" and hence should be exercised. All these rules are continuously evaluated before every event is enabled. If multiple cases are true, dependency has higher precedence than independency.

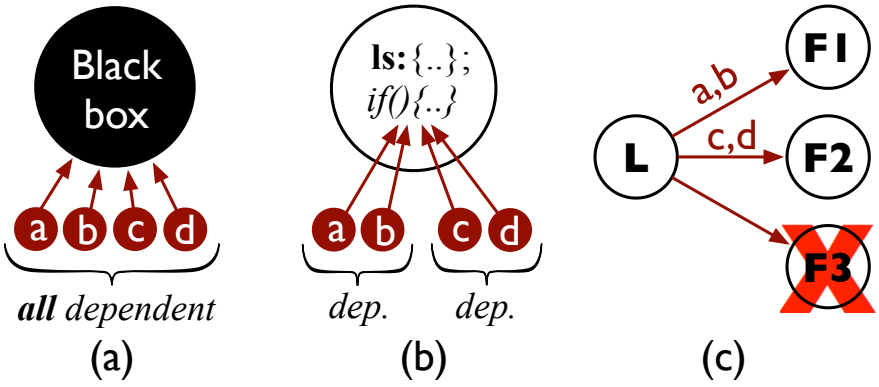

Figure 4.3: **LMI and CMI.** *The figures illustrate (a) a black-box approach, (b) local-message independence with white-box knowledge, and (c) crash-message independence.*

Overall, LMI allows dmck to smartly skip redundant re-orderings by leveraging simple patterns. The job of the tester is to find the message processing patterns from a target protocol and write *protocol-specific rules* (*i.e.*, filling in the content of the four LMI predicate functions (`pd`, `pi`, `pc`, and `pm`) specific to the target protocol). As an example, for our simple leader election protocol (Section **??**), `pd` can be as simple as: return `m.vote < ls.myVote`.

Crash-Message Independence (CMI)

Figure 4.3c illustrates the motivation behind our next policy. The figure resembles an atomic broadcast protocol where a leader (`L`) sends commit messages to the followers (`Fs`). Let's assume commit messages `ab` to `F1` and `cd` to `F2` are still in flight (*i.e.*, currently outstanding in the dmck; not shown). In addition, the dmck would like to crash `F3`, which we label as a crash event `X`. The question we raise is: how should `X` be re-ordered with respect to other outstanding messages (`a`, `b`, `c`, and `d`)?

As we mentioned before, we find *no* single dmck that incorporates crash semantics into reduction policies. As an implication, in our example, the dmck must reorder `X` with respect to other outstanding messages, generating executions `Xabcd`, `aXbcd`, `abXcd`, and so on. Worse, when `abcd` are reordered, `X` will be reordered again. We find this as one major fundamental problem why existing dmcks do not scale with the inclusion of failures.

To solve this, we introduce crash-message independence (CMI) which defines *independency relationship between a to-be-injected crash and outstanding messages*. The key lies in these two crash reaction patterns (global vs. local impact) running on the surviving nodes (*e.g.*, the leader node in Figure 4.3c).

```
Global impact:          Local impact:
if (pg(X,ls))            if (pl(X,ls))
  modify(ls);              modify(ls);
  sendMsg();
```

The functions with prefix `p` are predicate functions that compare the crash event `X` with respect to the surviving node's local state (*e.g.*, the leader's local state). The `pg` predicate in the *global-impact* pattern defines that the crash `X` during the local state `ls` will lead to a local state change *and* new outgoing messages (*e.g.*, to other surviving nodes). Here, no reduction can be done because the new crash-induced outgoing messages must be re-ordered with the current outstanding messages. On the other hand, reduction opportunities exist within the *local-impact* pattern, wherein the `pl` predicate specifies that the crash will just lead to a local state change but not new messages, which implies that the crash does not need to be re-ordered.

Based on the two crash impact patterns, we apply CMI in the following ways. Given a local state `ls` at node `N`, a peer failure `X`, and outstanding messages ($\texttt{m}_1$...$\texttt{m}_n$) from `N` to other surviving peers, CMI performs: (1) If `pl` is true, then `X` and $\texttt{m}_1$...$\texttt{m}_n$ are independent. (2) If `pg` is true, then `X` and $\texttt{m}_1$...$\texttt{m}_n$ are dependent. In Figure 4.3c for example, if `pl` is true in node `L`, then `X` does not impact outstanding messages to `F1` and `F2`, and thus `X` is independent to `abcd`; exercising `Xabcd` is sufficient.

An example of CMI application is a quorum-based write protocol. If a follower crash occurs and quorum is still established, the leader will just decrease the number of followers (local state change only). Here, for the protocol-specific rules, the tester can specify `pl` with `#follower >= majority` and `pg` with the reverse. Overall, CMI helps dmck scale with the inclusion of failures,

```
broadcast()  sendMsgToAll(role, leaderId);
quorumOkay()  return (follower > nodes / 2);

// pr1
if (role == L && C.role == F && quorumOkay())
  follower--;

// pr2
if (role == L && C.role == F && !quorumOkay())
  follower = 0;
  role = S;
  broadcast();

// pr3
if (role == F && C.role == L)
  leaderId = myId;
  broadcast();
```

Figure 4.4: **Crash Recovery in Leader Election.** *The figure shows a simplified example of crash recovery in a leader election protocol. The code runs in every node.* `C` *implies the crashing node; each node typically has a view of the states of its peers. Three predicate-recovery pairs are shown (*$\texttt{pr}_1$*,* $\texttt{pr}_2$*, and* $\texttt{pr}_3$*). In the first, if quorum still exists, the leader simply decrements the follower count. In the second, if quorum breaks, the leader falls back to searching mode (*`S`*). In the third, if the leader crashes, the node (as a follower) votes for itself and broadcasts the vote to elect a new leader.*

specifically by skipping redundant re-orderings of crashes with respect to outstanding messages.

Crash Recovery Symmetry (CRS)

Before we discuss our next reduction policy, we emphasize again the difference between message event and crash/reboot event. Message events are generated by the target system, and thus dmck can only reduce the number of re-orderings (but it cannot reduce the events). Contrary, crash events are generated by dmck, and thus there exists opportunities to reduce the number of injected crashes. For example, in Figure 4.3c, in addition to crashing `F3`, the dmck can also crash `F1` and `F2` in different executions, but that might not be necessary.

To omit redundant crashes, we develop crash recovery symmetry (CRS). The intuition is that

some crashes often lead to symmetrical recovery behaviors. For example, let's assume a 4-node system with node roles `FFFL`. At this state, crashing the first or second or third node perhaps lead to the same recovery since all of them are followers, and thereby injecting one follower crash could be enough. Further on, if the system enters a slightly different state, `FFLF`, crashing any of the followers might give the same result as above. However, crashing the leader in the two cases (`N4` in the first case and `N3` in the second) should perhaps be treated differently because the recovery might involve the dead leader ID. The goal of CRS is to help dmck with crash decision.

The main question in implementing CRS is: how to incorporate crash recovery semantics into dmck? Our solution is to compute *recovery abstract global state* (`rags`), a simple and concise representation of crash recovery. CRS builds `rags` with the following steps:

First, we define that two recovery actions are symmetrical if they produce the same messages and change the same local states in the same way.

Second, we extract recovery logic from the code by flattening the predicate-recovery pairs (*i.e.*, recovery-related `if` blocks). Figure 4.4 shows a simple example. Different recovery actions will be triggered based on which recovery predicate ($\texttt{pr}_1$, $\texttt{pr}_2$, or $\texttt{pr}_3$) is true. Each predicate depends on the local state and the information about the crashing node. Our key here is to map each predicate-recovery pair to this formal pattern:

```
if (pr_i(ls, C.ls))
   modify(rals_i);
   (and/or)
   sendMsg(rals_i);
```

Here, $\texttt{pr}_i$ is the recovery predicate for the i-th recovery action, and $\texttt{rals}_i$ is the recovery abstract local state (*i.e.*, a subset of all fields of the local state involved in recovery). That is, each recovery predicate defines what recovery abstract local state that matters (*i.e.*, $\texttt{pr}_i \rightarrow \{\texttt{rals}_i\}$). For example, in Figure 4.4, if $\texttt{pr}_1$ is true, then $\texttt{rals}_1$ only contains the `follower` variable; if $\texttt{pr}_3$ is true, $\texttt{rals}_3$ contains `role` and `leaderId` variables.

Third, before we crash a node, we check which $\texttt{pr}_i$ will be true on each surviving node and then obtain the $\texttt{rals}_i$. Next, we combine $\texttt{rals}_i$ of all surviving nodes and *sort* them into a recovery abstract global state (`rags`); sorting `rags` helps us exploit topological symmetry (*e.g.*, individual node IDs often do not matter).

Fourth, given a plan to crash a node, the algorithm above gives us the `rags` that represents the corresponding recovery action. We also maintain a history of `rags` of previous injected crashes. If the `rags` already exists in the history, then the crash is skipped because it will lead to a symmetrical recovery of the past.

To recap with a concrete example, let's go back to the case of `FFFL` where we plan to enable crash(`N1`). Based on the code in Figure 4.4, the `rags` is {*, ⊘, ⊘, `#follower=3`}; * implies the crashing node, ⊘ means there is no true predicate at the other two follower nodes, and `#follower=3` comes from $\texttt{rals}_1$ of $\texttt{pr}_1$ of `N4` (the leader). CRS will sort this and check the history, and assuming no hit, then crash(`N1`) will be enabled. In another execution, SAMC finds that crash(`N2`) at `FFFL` will lead to `rags`:{⊘, *, ⊘, `#follower=3`}, which after sorting will hit the history, and hence crash(`N2`) is skipped. If the system enters a different state `FFLF`, no follower crash will be injected, because the `rags` will be the same as above. In terms of leader crash, crashing the leader in the two cases will be treated differently because in a leader crash, $\texttt{pr}_3$ is true on followers and $\texttt{pr}_3$ involves `leaderId` which is different in the two cases.

In summary, the foundation of CRS is the computation of recovery abstract global state (`rags`) from the crash recovery logic extracted from the target system via the $\texttt{pr}_i \rightarrow \{\texttt{rals}_i\}$ pattern. We believe this extraction method is simple because CRS does *not* need to know the specifics of crash recovery; CRS just needs to know what variables are involved in recovery (*i.e.*, the `rals`) .

Reboot Synchronization Symmetry (RSS)

Reboots are also essential to exercise (Section **??**), but if not done carefully, will introduce more scalability problems. Reboot reduction policy is needed to help dmck inject reboots "smartly".

The intuition behind reboot synchronization symmetry (RSS) is similar to that of CRS. When a node reboots, it typically *synchronizes* itself with the peers. However, a reboot will not lead to a new scenario if the current state of the system is similar to the state when the node crashed. To implement RSS, we extract reboot-synchronization predicates and the corresponding actions. Since the overall approach is similar to CRS, we omit further details.

In our experience RSS is extremely powerful. For example, it allows us to find deep bugs involving multiple reboots in the ZooKeeper atomic broadcast (ZAB) protocol. RSS works efficiently here because reboots in ZAB are only interesting if the live nodes have seen new commits (*i.e.*, the dead node falls behind). In contrast, a black-box dmck without RSS initiates reboots even when the live nodes are in similar states as in before the crash, prolonging the discovery of deep bugs.

4.2.4 Pattern Extraction

We have presented four general, simple, and powerful semantic-aware reduction policies along with the generic event processing patterns. With this, testers can write protocol-specific rules by extracting the patterns from their target systems. Given the patterns described in previous sections, a tester must perform what we call as "extraction" phase. Here, the tester must extract the patterns from the target system and write protocol-specific rules specifically by filling in the predicates and abstractions as defined in previous sections; in Section 4.3.2, we will show a real extraction result (*i.e.*, real rules). Currently, the extraction phase is manual; we leave automated approaches as a future work (Section 4.5). Nevertheless, we believe manual extraction is bearable for several reasons. First, today is the era of DevOps [89] where developers are testers and vice versa; testers know the internals of their target systems. This is also largely true in cloud system development. Second, the processing patterns only cover high-level semantics; testers just fill in the predicates and abstractions but no more details. In fact, simple semantics are enough to significantly help dmck go faster to deeper states.

4.3 Implementation and Integration

In this section, we first describe our SAMC prototype, SAMPRO, which we built from scratch because existing dmcks are either proprietary [126] or only work on restricted high-level languages (*e.g.*, Mace [82]). We will then describe SAMPRO integration to three widely popular cloud systems, ZooKeeper [73], Hadoop/Yarn [118], and Cassandra [84]. Prior to SAMPRO, there was no available dmck for these systems; they are still tested via unit tests, and the test code size is bigger than the main code, but the tests are far from reaching deep bugs.

4.3.1 SAMPRO

SAMPRO is written in 10,886 lines of code in Java, which includes all the components mentioned in Section 4.1.1 and Figure 4.1. The detailed anatomy of dmck has been thoroughly explained in literature [60, 63, 82, 114, 126], and therefore for brevity, we will not discuss many engineering details. We will focus on SAMC-related parts.

We design SAMPRO to be highly portable; we do not modify the target code base significantly as we leverage a mature interposition technology, AspectJ, for interposing network messages and timeouts. Our interposition layer also sends local state information to the SAMPRO server. SAMPRO is also equipped with crash and reboot scripts specific to the target systems. The tester can specify a budget of the maximum number of crashes and reboots to inject per execution. SAMPRO employs basic reduction mechanisms and advanced reduction policies as described before. We deploy safety checks at the server (*e.g.*, no two leaders). If a check is violated, the trace that led to the bug is reported and can be deterministically replayed in SAMPRO. Overall, we have built all the necessary features to show the case of SAMC. Other features such as intra-node thread interleavings [63], scale-out parallelism [115], and virtual clock for network delay [126] can be integrated to SAMPRO as well.

4.3.2 *Integration to Target Systems*

In our work, the target systems are ZooKeeper, Hadoop 2.0/Yarn, and Cassandra. ZooKeeper [73] is a distributed synchronization service acting as a backbone of many distributed systems such as HBase and High-Availability HDFS. Hadoop 2.0/Yarn [118] is the current generation of Hadoop that separates cluster management and processing components. Cassandra [84] is a distributed key-value store derived from Amazon Dynamo [45].

In total, we have model checked 7 protocols: ZooKeeper leader election (ZLE) and atomic broadcast (ZAB), Hadoop cluster management (CM) and speculative execution (SE), and Cassandra read/write (RW), hinted handoff (HH) and gossiper (GS). These protocols are highly asynchronous and thus susceptible to message re-orderings and failures.

Table 4.1 shows a real sample of protocol-specific rules that we wrote. Rules are in general very short; we only wrote 35 lines/protocol on average. This shows the simplicity of SAMC's integration to a wide variety of distributed system protocols.

4.4 Evaluation

We now evaluate SAMC by presenting experimental results that answer the following questions:

1. How fast is SAMC in finding deep bugs compared to other state-of-the-art techniques?
2. Can SAMC find new deep bugs?
3. How much reduction ratio does SAMC provide?

To answer the first question, we show SAMC's effectiveness in finding old bugs. For this, we have integrated SAMPRO to old versions of our target systems that carry deep bugs: ZooKeeper v3.1.0, v3.3.2, v3.4.3, and v3.4.5, Hadoop v2.0.3 and v2.2.0, and Cassandra v1.0.1 and v1.0.6. To answer the second question, we have integrated SAMPRO to two recent stable versions: ZooKeeper v3.4.6 (released March 2014) and Hadoop v2.4.0 (released April 2014). In total, we have integrated

Local-Message Independence (LMI)

```
bool pd : !newVote(m, s);

bool pm : newVote(m, s);

bool newVote(m, s) :
 if (m.ep > s.ep)
   return true;
 else if (m.ep == s.ep)
  if (m.tx > s.tx)
   return true;
  else if (m.tx == s.tx &&
           m.lid > s.lid)
   return true;
```

Crash-Message Independence (CMI)

```
bool pg (s, X) :
 if (s.rl == F && X.rl == L)
  return true;
 if (s.rl == L && X.rl == F
     && !quorumAfterX(s)
  return true;
 if (s.rl == S && X.rl == S)
  return true;

bool pl (s, X) :
 if (s.rl == L && X.rl == F
     && quorumAfterX(s))
  return true;

bool quorumAfterX(s) :
  ret ((s.fol-1) >=
        s.all/2);
```

Crash Recovery Symmetry (CRS)

```
bool pr1(s,C):
 if (s.rl == L && C.rl == F
     && quorumAfterX(s))
  return true;
rals1:{rl,fol,all};

bool pr2(s,C):
 if (s.rl == L && C.rl == F
     && !quorumAfterX(s))
 return true;
rals2: {rl,fol,lid,ep,tx,clk}

bool pr3(s,C):
 if (s.rl == F && c.rl == L)
  return true;
rals3: {rl,fol,lid,ep,tx,clk}

bool pr4:
 if (s.rl == S)
  return true;
rals4: {rl,lid,ep,tx,clk}
```

Reboot Synchronization Symmetry (RSS)

```
bool ps1(s,R):
 if (s.rl == L)
  return true;
sals1: {rl,lid,ep,tx,clk}

bool ps2(s,R):
 if (s.rl == F)
  return true;
sals2: {rl,lid,ep,tx,clk}

bool ps3(s,R):
 if (s.rl == S &&
     s.clk > R.clk)
  return true;
sals3: {rl,lid,ep,tx,clk}

bool ps4(s,R):
 if (s.rl == S &&
     moreUpdated(s, R))
  return true;
sals4: {rl,lid,ep,tx,clk}

bool moreUpdated(s, R):
 if (R.ep > s.ep)
  return true;
 else if (R.ep == s.ep)
  if (R.tx > s.tx)
   return true;
  else if (R.tx == s.tx)
   if (R.lid > s.lid)
    return true;
```

Table 4.1: **Protocol-Specific Reduction Rules for ZLE.** *The code above shows the actual protocol-specific rules for ZLE protocol. These rules are the inputs to the four reduction policies. Many variables are abbreviated (ep: epoch, tx: latest transaction ID, lid: leader ID, rl: role, fol: follower count, all: total node count, clk: logical clock, L: leading, F: following, S: searching, X/C: crashing node, R: rebooting node). LMI* `pc` *and* `pi` *predicates are not used for ZLE, but used for other protocols.*

SAMPRO to 10 versions, showing the high portability of our prototype. Overall, our extensive evaluation exercised more than 100,000 executions and used approximately 48 full machine days.

4.4.1 Speed in Finding Old Bugs

This section evaluates the speed of SAMC vs. state-of-the-art techniques in finding old deep bugs. In total, we have reproduced 12 old deep bugs (7 in ZooKeeper, 3 in Hadoop, and 2 in Cassandra). Figure 4.5 illustrates the complexity of the deep bugs that we reproduced.

					#Executions				Speed-up of SAMC vs.		
Old Issue#	**Protocol**	**E**	**C**	**R**	**bDP**	**RND**	**rDP**	**SAMC**	**bDP**	**RND**	**rDP**
ZK-335	ZAB	120	3	3	↑5000	1057	↑5000	117	⇑43	9	⇑43
ZK-790	ZLE	21	1	1	14	225	82	7	2	32	12
ZK-975	ZLE	21	1	1	967	71	163	53	18	1	3
ZK-1075	ZLE	25	3	2	1081	86	250	16	68	5	16
ZK-1419	ZLE	25	3	2	924	2514	987	100	9	25	10
ZK-1492	ZLE	31	1	0	↑5000	↑5000	↑5000	576	⇑9	⇑9	⇑9
ZK-1653	ZAB	60	1	1	945	3756	3462	11	86	341	315
MR-4748	SE	25	1	0	22	6	6	4	6	2	2
MR-5489	CM	20	2	1	↑5000	↑5000	↑5000	53	⇑94	⇑94	⇑94
MR-5505	CM	40	1	1	1212	↑5000	1210	40	30	⇑125	30
CA-3395	RW+HH	25	1	1	2552	191	550	104	25	2	5
CA-3626	GS	15	2	1	↑5000	↑5000	↑5000	96	⇑52	⇑52	⇑52

Table 4.2: **SAMC Speed in Finding Old Bugs.** *"E", "C" and "R" represent the number of events, crashes, and reboots necessary to hit the bug. The numbers in the middle four columns represent the number of executions to hit the bug across different policies. "bDP", "RND", and "rDP" stand for black-box DPOR (in* MODIST*), random, and random + black-box DPOR respectively. We stop at 5000 executions (around 2 days) if the bug cannot be found (labeled with "↑"). Thus, speed-up numbers marked with "⇑" are potentially much higher.*

Table 4.2 shows the result of our comparison. We compare SAMC with basic techniques (DFS and Random) and advanced state-of-the-art techniques such as black-box DPOR ("bDP") and Random+bDP ("rDP"). Black-box DPOR is the MODIST-style of DPOR (Section 6.1). We include Random+DPOR to mimic the way MODIST authors found bugs faster (Section 4.1.3). The table shows the number of executions to hit the bug. As a note, software model checking with the inclusion of failures takes time (back-and-forth communications between the target system and the

MR-5505:

1. A job finishes,
2. Application manager (AM) sends a "remove-app" message to Resource Manager (RM),
3. RM receives the message,
4. AM is unregistering,
5. *RM crashes* before completely processes the message,
6. AM finishes unregistering,
7. *RM reboots* and reads the old state file,
8. RM thinks the job has never started and runs the job again.

CA-3395

1. Three nodes N1-3 started and formed a ring,
2. Client writes data,
3. *N3 crashes*,
4. Client updates the data via N1; N3 misses the update,
5. *N3 reboots*,
6. N1 begins the hinted handoff process, (7) Another client reads the data with strong consistency via N1 as a coordinator,
7. N1 and N2 provide the updated value, but N3 still provides the stale value,
8. The coordinator gets "confused" and returns the stale value to the client!

Figure 4.5: **Complexity of Deep Bugs.** *Above are two sample deep bugs in Hadoop and Cassandra. A sample for ZooKeeper was shown in the introduction (Section* **??***). Deep bugs are complex to reproduce; crash and reboot events must happen in a specific order within a long sequence of events (there are more events behind the events we show in the bug descriptions above). To see the high degree of complexity of other old bugs that we reproduced, interested readers can click the issue numbers (hyperlinks) in Table 4.2.*

dmck server, killing and restarting system processes multiple times, restarting the whole system from a clean state, etc.). On average, each execution runs for 40 seconds and involves a long sequence of 20-120 events including the necessary crashes and reboots to hit the bug. We do not show the result of running DFS because it never hits most of the bugs.

Based on the result in Table 4.2, we make several conclusions. First, with SAMC, we prove that smart systematic approaches can reach to deep bugs quickly. We do not need to revert to randomness or incorporate checkpoints. As a note, we are able to reproduce every deep bug that we picked; we did not skip any of them. (Hunting more deep bugs is possible, if needed).

Second, SAMC is one to two orders of magnitude faster compared to state-of-the-art techniques. Our speed-up is up to 271x (33x on average). But most importantly, there are bugs that other techniques cannot find even after 5000 executions (around 2 days). Here, SAMC's speed-up factor is potentially much higher (labeled with "⇑"). Again, in the context of dmck (a process of hours/days), large speed-ups matter. In many cases, state-of-the-art policies such as bDP and rDP cannot reach the bugs even after very long executions. The reasons are the two problems we mentioned earlier (Section 4.1.3). Our micro-analysis (not shown) confirmed our hypothesis that non-SAMC policies frequently make redundant crash/reboot injections and event re-orderings that anyway lead to insignificant state changes.

Third, Random is truly "random". Although many previous dmcks embrace randomness in finding bugs [82, 126], when it comes to failure-induced bugs, we have a different experience. Sometimes Random is as competitive as SAMC (*e.g.*, ZK-975), but sometimes Random is much slower (*e.g.*, ZK-1419), or worse Random sometimes did not hit the bug (*e.g.*, ZK-1492, MR-5505). We find that some bugs require crashes and/or reboots to happen at very specific points, which is probabilistically hard to reach with randomness. With SAMC, we show that being systematic and semantic aware is consistently effective.

4.4.2 *Ability of Finding New Bugs*

The previous section was our main focus of evaluation. In addition to this, we have integrated SAMPRO to recent stable versions of ZooKeeper (v3.4.6, released March 2014) and Hadoop (v2.4.0, released April 2014). In just hours of deployment, we found 1 new ZLE bug involving 2 crashes, 2 reboots, and 52 events, and 1 new Hadoop speculative execution bug involving 2 failures and 32 events. These two new bugs are distributed data race bugs. The ZLE bug causes the ZooKeeper cluster to create two leaders at the same time. The Hadoop bug causes a speculative attempt on a job that is wrongly moved to a scheduled state, which then leads to an exception and a failed job. We can deterministically reproduce the bugs multiple times and we have reported the bugs to the developers. Currently, the bugs are still marked as major and critical, the status is still open, and the resolution is still unresolved.

We also note that in order to unearth more bugs, a dmck must have several complete features: workload generators that cover many protocols, sophisticated perturbations (*e.g.*, message re-ordering, fault injections) and detailed checks of specification violations. Further discussions can be found in our previous work [61]. Currently, SAMPRO focuses on speeding up the perturbation part. By deploying more workload generators and specification checks in SAMPRO, more deep bugs are likely to be found. As an illustration, the 94 deep bugs we mentioned in Section **??** originated from various protocols and violated a wide range of specifications.

4.4.3 *Reduction Ratio*

Table 4.3 compares the reduction ratio of SAMC over black-box DPOR (bDP) with different budgets (#crashes and #reboots). This evaluation is slightly different than the bug-finding speed evaluation in Section 4.4.1. Here, we measure how many executions in bDP are considered redundant based on our reduction policies and protocol-specific rules. Specifically, we run bDP for 3000 executions and run SAMC policies on the side to mark the redundant executions. The reduction ratio is then 3000 divided by the number of non-redundant executions. Table 4.3 shows that SAMC

C	R	Execution Reduction Ratio in ZLE	Execution Reduction Ratio in ZAB
1	1	37	93
2	2	63	107
3	3	103	166

(a)

C	R	Execution Reduction Ratio in ZLE with All	LMI	CMI	CRS	RSS
1	1	37	18	5	4	3
2	2	63	35	6	5	5
3	3	103	37	9	9	14

(b)

Table 4.3: **SAMC Reduction Ratio.** *The table (a) shows the execution reduction ratio of SAMC over black-box DPOR (bDP) in checking ZLE and ZAB under different crash/reboot budgets. "C" and "R" are the number of crashes and reboots. The table (b) shows the execution reduction ratio in ZLE with individiual policies over black-box DPOR (bDP).*

provides between 37x-166x execution reduction ratio in model checking ZLE and ZAB protocols across different crash/reboot budgets.

Table 4.3b shows that with each policy the execution reduction ratio increases when the number of crashes and reboots increases. With more crashes and reboots, the ZLE protocol generates more messages and most of them are independent, and thus the LMI policy has more opportunities to remove redundant message re-orderings. Similary, the crash and reboot symmetry policies give better benefits with more crashes and reboots. The table also shows that LMI provides the most reduction. This is because the number of message events is higher than crash and reboot events (as also depicted in Table 4.2).

We now discuss our reduction ratio with that of DIR [63]. As discussed earlier (Section 6.1), DIR records local exploration (thread interleavings) and replays future incoming messages whenever possible, reducing the work of global exploration. If the target system does not have lots of thread interleavings, DIR's reduction ratio is estimated to be between 10^1 to 10^3 (Section 5 of [63]). As we described earlier (Section 6.1), DIR is orthogonal to SAMC. Thus, the reduction ratios of SAMC and DIR are complementary; when both methods are combined, there is a potential for a higher reduction ratio. The DIR authors also hinted that domain knowledge can guide dmcks (and also help their work) to both scale and hit deep bugs (Section 8 of [63]). SAMC has successfully addressed such need.

Finally, we note that in evaluating SAMC, we use execution reduction ratio as a primary metric.

Another classical metric to evaluate a model checker is state coverage (*e.g.*, a dmck that covers more states can be considered a more powerful dmck). However, in our observation state coverage is not a proper metric for evaluating optimization heuristics such as SAMC policies. For example, if there are three nodes ABC that have the same role (*e.g.*, follower), a naive black-box dmck will crash each node and covers three distinct states: *BC, A*C and AB*. However, with a semantic-aware approach (*e.g.*, symmetry), we know that covering one of the states is sufficient. Thus, less state coverage does not necessarily imply a less powerful dmck.

4.5 Discussion

In this section, we discuss SAMC's simplicity, generality and soundness. We would like to emphasize that the main goal of this thesis is to show the power of SAMC in finding deep bugs both quickly and systematically, and thus we intentionally leave some subtasks (*e.g.*, automated extraction, soundness proofs) for future work.

4.5.1 Simplicity

In previous sections, we mentioned that policies can be written in few lines of code. Besides LOC, simplicity can be measured by how much time is required to understand a protocol implementation, extract the patterns and write the policies. This time metric is unfortunately hard to quantify. In our experience, the bulk of our time was spent in developing SAMPRO from scratch and integrating policies to dmck mechanisms (Section 4.1.1). However, the process of understanding protocols and crafting policies requires a small effort (*e.g.*, few days per protocol to the point where we feel the policies are robust). We believe that the actual developers will be able to perform this process much faster than we did as they already have deeper understandings of their code.

4.5.2 *Generality*

Our policies contain patterns that are common in distributed systems. One natural question to ask is: how much semantic knowledge should we expose to dmck? The ideal case is to expose as much information as possible as long as it is sound. Since proving soundness and extracting patterns automatically are our future work, in this thesis we only propose exposing high-level processing semantics. With advanced program analysis tools that can analyze deep program logic, we believe more semantic knowledge can be exposed to dmck in a sound manner. For example, LMI can be extended to include commutative modifications. This is possible if the program analysis can verify that the individual modification does not lead to other state changes. This will perhaps be the point where symbolic execution and dmck blend in the future (Section **??**).

Nevertheless, we find that high-level semantics are powerful enough. Beyond the three cloud systems we target in this thesis, we have been integrating SAMC to MaceMC [82]. MACEMC already employs random exploration policies to model check Mace-based distributed systems such as Mace-based Chord and Pastry. To integrate SAMC, we first must re-implement DPOR in MaceMC (existing DPOR implementation in MaceMC is proprietary [63]). Then, we have written 18 lines of LMI protocol-specific rules for Chord and attain two orders of magnitude of reduction in execution. This shows the generality of SAMC to many distributed systems.

4.5.3 *Soundness*

SAMC policies only skip re-orderings and crash/reboot events that are redundant by definition, however currently our version of SAMC is not sound; the unsound phase is the manual extraction process. For example, if the tester writes a wrong predicate definition (*e.g.*, `pd`) that is inconsistent with what the target system defines, then soundness (and correctness) is broken. Advanced program analysis tools can be developed to automate and verify this extraction process and make SAMC sound. Currently, the fact that protocol-specific rules tend to be short might also help in reducing human errors. Our prototype, SAMPRO, is no different than other testing/verification

tools; full correctness requires that such tools to be free of bugs and complete in checking all specifications, which can be hard to achieve. Nevertheless, we want to bring up again the discussion in Section 4.1.3 that dmck's scalability and ability to find deep bugs in complex distributed systems are sometimes more important than soundness. We leave soundness proofs for future work, but we view this as a small limitation, mainly because we have successfully shown the power of SAMC.

CHAPTER 5

A CASE OF SCALABILITY BUG IN CLOUD-SCALE DISTRIBUTED SYSTEMS

This chapter highlights an urgency in tackling scalability bugs by studying deeply in 41 scalability bugs from different popular scalable distributed systems. Section 5.1 discusses motivation in tackling scalability bugs and Section 5.2 gives some observations we gain from the study.

5.1 Motivation

Is scale a friend or a foe [103]? On the positive side, scale surpasses the limit of a single machine in meeting users' increasing demands of compute and storage, which led to many inventions of "cloud-scale" distributed systems [38, 44, 45, 58, 70, 120]. The field has witnessed a phenomenal deployment scale of such systems; Netflix runs tens of 500-node Cassandra clusters [31], Apple deploys a total of 100,000 Cassandra nodes [4], and Yahoo! recently revealed the use of 40,000 Hadoop servers, with a 4500-node cluster as the largest one [32].

On the negative side, scale creates new development and deployment issues. Developers must ensure that their algorithms and protocol designs to be scalable. However, until real deployment takes place, unexpected bugs in the actual implementations are unforeseen. This new era of cloud-scale distributed systems has given birth to a new type of bug: *scalability bugs*. They are latent bugs that are scale-dependent; they only surface in large-scale deployments, but not in small/medium-scale ones. Their presence jeopardizes systems reliability and availability at scale.

As an example, let us consider a bug in Cassandra, a highly-scalable peer-to-peer key-value store. If a customer initially deploys a cluster of 50 nodes and later scales it out with 50 additional nodes, the operation can be done smoothly. However, if the customer deploys a 200-node cluster and then adds 200 more nodes, the protocol that rebalances the key-range partitions (which nodes should own which key ranges) becomes CPU intensive as the calculation has an $O(N^3)$ complexity

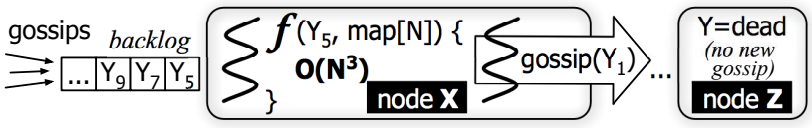

Figure 5.1: **Cassandra bug CA-6127.** *The problem of gossip-based failure detection in Cassandra*

where N is the number of nodes. This combined with the gossiping and failure detection logic leads to a scalability bug that makes the cluster unstable (many live nodes are declared as dead, making some data not reachable by the users). We give full detail of the Cassandra bug in Section 5.1.1.

We perform an in-depth study of 41 scalability bugs reported from the deployments of popular large-scale systems such as Hadoop, HBase, HDFS, Cassandra, Couchbase, Riak, and Voldemort. From this study, we observed many challenges in finding, reproducing, and debugging scalability bugs. As in the example above, bug symptoms sometimes surface only in large deployment scales (*e.g.*, N>100 nodes), hence small/medium-scale testing is not enough. Yet, not all developers have large test budgets, and even when they do, debugging on hundreds of nodes is time consuming and difficult. Furthermore, protocol algorithms can be scalable in the design sketches, but not necessarily in the real deployments; there are specific implementation details whose implications at scale are hard to predict. We discuss more about our observations on scalability bugs in Section 5.2.

5.1.1 A Sample Cassandra Bug

We now describe in detail a scalability bug in Cassandra, which we use as a sample bug. Our journey in understanding scalability bugs began when we observed repeated "flapping" problems in large-scale Cassandra deployments (*i.e.*, hundreds of nodes). Flapping is a cluster instability problem where node's up/down status continuously flaps. A "flap" is when a node X marks a peer node Y as down (and soon marks Y as alive again). We rigorously study a series of Cassandra bugs below that surfaced as the code evolved.

To understand this bug, we need to understand the following protocols.

1. **Bootstrap:** Each node first creates partition keys (*e.g.*, 32 random numbers) and gossips this information to peer nodes.

2. **Gossip broadcast:** *Every second*, each node gossips to one random node about a list of nodes and partitions it knows (including itself) and their *version* numbers. Each node also increments its version number ("I'm still alive") before gossiping.

3. **Gossip processing:** The receiving node then finds any state (metadata) differences between the two nodes to synchronize their views of the ring. Eventually, all nodes know about each other.

4. **Failure detection:** *Every second*, a failure detection daemon runs [84]. Put simply, if a node X has not received a new gossip about Y *from anyone* (Y's version has not changed after some period of time), X will declare Y dead (a flap). When X receives a new gossip about Y, it marks Y alive.

There are two factors that induce the bug. The first is the *long latency of scale-dependent state-update gossip processing during bootstrapping* ("f" in Figure 5.1). While gossip processing is usually fast in a stable cluster, it is expensive during bootstrapping as the gossips carry many new state changes about the ring; the state-update processing time is scale-dependent (*i.e.*, greater than $O(N^3)$); the larger the cluster (N), the larger the ring map, the longer the processing time is. This long latency is caused by **(1)** state-update checkpoint to on-disk database and **(2)** multi-map cloning and updates. The first one is needed for fast fault tolerance; after a node crashes, it can reboot fast as it knows the latest view of the ring. The second one is preferred for simplicity; Cassandra clones its `MultiMap` ring table and applies changes one by one to alleviate long write locks.

The second factor is the *single threaded* implementation of gossip processing. As shown in Figure 5.1, this inability to process multiple gossips/state updates concurrently (for the sake of

preventing concurrency bugs) creates a *backlog* of new gossips. For example, in *every second*, Y tells someone it's alive with increasing version number (*e.g.*, Y_7), but the receiving nodes are still busy processing state changes and only forward Y's old version number (*e.g.*, Y_1). As Y's new gossip is not propagated on time, other nodes (*e.g.*, Z) will mark Y as dead. This happens to all nodes, not just Y.

The journey starts with Bug #CA-3831 [10], when a node D is decommissioned from a cluster ring, D initiates a gossip telling that all other nodes must rebalance the ring's key-ranges. This scale-dependent "pending key-range calculation" is CPU intensive with $O(MN^3log^3(N))$ complexity; M is the list of key-range changes in the gossip message. This in turn leaves many gossips not propagated on time, creating flapping symptoms that only appear at scale (at 200+ nodes; Section 6.3). The developers then optimized the code to $O(MN^2log^2(N))$ complexity.

Soon afterwards (Bug #CA-3881 [11]), Cassandra added the concept of virtual partitions/nodes (*e.g.*, P=256 per physical node). As an implication, the fix above did not scale as "N" becomes $N \times P$. The bug was fixed with a complete redesign of the pending key-range calculation, making it $O(MNPlog^2(NP))$.

About a year later (CA-5456 [12]), Cassandra code employs multi-threading between the pending key-range calculation and the gossip processing with a coarse-grained lock to protect sharing of the ring table. Unbeknownst to the developers, at scale, the key-range calculation can acquire the lock for a long time, causing flapping to reappear again. The fix clones the ring table for the key-range calculation, to release the lock early.

Later on (CA-6127 [13]), a similar bug reappeared. In the above cases, the problems appeared when the cluster grows/shrinks gradually. However, if customers bootstrap a large cluster (*e.g.*, 500+ nodes) from scratch (*i.e.*, all nodes do not know each other, with no established key ranges), the execution traverses a different code path that performs a fresh ring-table/key-range construction with $O(MN^2)$ complexity.

The story continues on (CA-6345, CA-6409, etc.). Fast forward today, Cassandra developers

recently started a new umbrella ticket for discussing "Gossip 2.0," supposedly scalable to 1000+ nodes [20, 21]. Similar to Cassandra, other large-scale systems are prone to the same problem. So far, we have collected and analyzed 9 Cassandra, 5 Couchbase, 2 Hadoop, 10 HBase, 13 HDFS, 1 Riak, and 1 Voldemort scalability bugs, all caused user-visible impacts. This manual mining was arduous because there is no searchable jargon for "scalability bugs"; we might have missed other bugs.

5.2 Observations

From the bug in previous section and all the bugs we studied, we make several important observations.

- *Only appear at extreme scale:* CA-6127 does not surface in 30-node deployment. In 128-node cluster, the symptom appears mildly (tens of flaps). From 200-500 nodes, flapping skyrockets from hundreds to thousands of flaps. Testing in small/medium scales is not sufficient, which is also true for other bugs we studied (Section 6.3).

- *Scalable in design, but not in practice.* Related to CA-6127, the accrual failure detector/gossiper [68] was interestingly adopted by Cassandra as it is scalable in design [84]. However, the design proof does not account gossip processing time during bootstrap, which can be long. To understand the bug, the developers tried to "do the [simple] math" [13] but failed. In practice, the assumption that new gossips are propagated every second is not met (due to the backlog). The actual implementations overload gossips with many other purposes (*e.g.*, announcing boot/rebalance changes) beyond their original design sketch.

- *Implementation specific and hard to predict.* The backlog-induced flapping in CA-6127 was caused specifically by Cassandra's implementation choice: metadata checkpoint, multi-map cloning, and its single-threaded implementation. State-update processing time is hard

to predict (ranges from 0.001 to 4 seconds) as it depends on a 2-dimensional input: the receiving node's ring table size and the number of new state changes (Section 6.3).

- *Cascading impacts of "not-so-independent" nodes.* In cluster-wide control protocols, distributed nodes are not necessarily independent; nodes must communicate with each other to synchronize their views of cluster metadata. As the cluster grows, the cluster metadata size increases. Thus, unpredictable processing time in individual nodes can create cascading impacts to the whole cluster.

- *Long and difficult large-scale debugging:* The bug report of CA-6127 generated over 40 back-and-forth discussion comments and took 2 months to fix. It is apparent [13] that there were many hurdles of deploying and debugging the buggy protocol at real scale. Important to note is that debugging is *not* a single iteration; developers must *repeatedly* instrument the system (add more logs) and re-run the system at scale to find and fix the bug, which is not trivial. The scalability bugs we studied took 6 to 157 days to fix (27 on average).

- *Not all developers have large test budgets:* Another factor of delayed fixes is the lack of budget for large test clusters. Such luxury tends to be accessible to developers in large companies, but not to open-source developers. When CA-6127 was submitted by a customer who had hundreds of nodes, the Cassandra developers did not have an instant access to a test cluster of the same scale.

- *Quick fixes and repeated bugs:* Bugs are often fixed with quick patches (development pressures), but the new fix might not eradicate the problem completely [128]. For example, for CA-6127, the patch simply disables failure detection during bootstrap. As the protocol was not redesigned, the bug still appeared in another workload (*e.g.*, scaling out from 128 to 256 nodes). In the latest Cassandra, the simple fix has been removed and the gossip protocol has been redesigned. We also found that old fixes can become obsolete in protocol re-designs, which then can give birth to new scalability bugs. For example, the fix for CA-3831 became

obsolete as “vnodes” was introduced, which then gave rise to a new vnode-related scalability bug (CA-3881). A scale-check could have ensured that new fixes remove old scalability bugs entirely and similar bugs do not re-surface in new designs.

CHAPTER 6

SCK: A SINGLE-MACHINE APPROACH FOR FINDING SCALABILITY BUGS IN CLOUD-SCALE DISTRIBUTED SYSTEMS

In the previous chapter, we discuss an urgency and motivation for testing scalability of cloud distributed systems. In this chapter, we discuss the-state-of-the-art techniques to test scalability and their limitations in Section 6.1, and propose SCK, a methodology to reveal scalability bugs in distributed systems economically by using only a single machine in Section 6.2.

6.1 State of the Art for Large-Scale Emulation

As we explain in Chapter 2 and show in Chapter 5, we need to check actual implement of the systems at real large scale, not simulation nor small scale setup, and that makes emulation approach as a desire choice. In this section, we will explore state of the art for large-scale emulation.

DieCast [64], invented for network emulation, can colocate many processes/VMs on a single machine as if they run individually without contention. The trick is adding "time dilation factor" (TDF) support [65] into the VMM (*e.g.*, Xen). For example, TDF=5 implies that for every second of wall-clock time, each emulated VM on the VMM believes that time has advanced by only 200 ms. The most significant drawback of DieCast is that high colocation factor (*e.g.*, TDF=100) is likely not desirable, for two reasons: prolonged testing time (TDF=100 implies 100x longer run) and memory overcapacity. DieCast was only evaluated with TDF=10.

Exalt [121] targets I/O-intensive scalability bugs. With a custom data compression, users' data is compressed to zero byte on disk (but the size is recorded) while metadata is not compressed. With this, Exalt can co-locate 100 emulated HDFS datanodes on one machine. In its evaluation, most of the bugs reproduced are in the HDFS namenode which runs alone on one machine. As the authors stated, their approach "may not discover scalability problems that arise at the nodes that are being emulated [the datanodes]" (Section 4.1 in [121]). Thus, Exalt is not suitable for finding

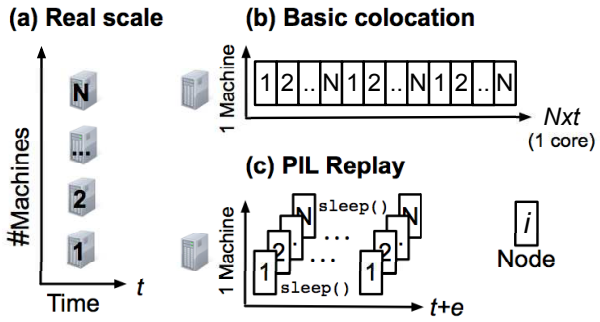

Figure 6.1: **Various scale-testing approaches.** *The left figure (a) illustrates a real-scale testing where the system/protocol under test is deployed on N machines, which illustratively takes t time to complete. The top figure (b) depicts a basic colocation where N nodes are packed into a single machine and exhibit CPU contention and context switching, which can take $N \times t$ time to complete (in one-processor scenario). The bottom figure (c) illustrates our processing illusion (PIL) as described in Section 6.2. Here, expensive functions are emulated with* `sleep()`*, thus the test time $t+e$ is similar to the real-scale testing.*

control-plane scalability bugs in P2P distributed systems.

In summary, we did not find a fast single-machine approach that can scale-check CPU-intensive protocols in cloud systems. The scalability bugs could be caused by the scale-dependent processing time, not network or I/O bottlenecks. As DieCast targets *network* emulation via time dilation and Exalt targets *storage* space emulation via compression.

6.2 SCK

We believe new methods are needed to help developers check their systems/protocols implementation at real scale but without the hurdles of running large test clusters. In our work, we explore a new approach to find and replay scalability bugs in a "cheap" way such as on one machine, which we name *single-machine scale check* (or just "scale check" short).

The research question to address is: how to colocate a large number of CPU-intensive nodes on one machine with limited resources and yet still achieve high accuracy? High accuracy implies that the colocated nodes generate a similar behavior as if they run on independent machines. The reason for inaccuracy is illustrated in Figures 6.1a and 6.1b. With real-scale testing (Figure 6.1a),

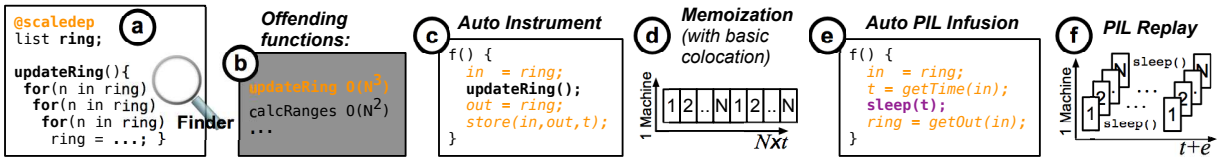

Figure 6.2: **The proposed flow of an automated scale-check process.**

the protocol under test might finish in t seconds. However, with a basic colocation, the CPU-intensive nodes contend with each other in one machine. With only just 1 processor core for example, the protocol under test might finish in $N \times t$ seconds, hence the inaccuracy.

To address this, below we present the concept of processing illusion (PIL) and how to find PIL-safe functions and generate output of PIL-replaced functions.

• **Processing Illusion (PIL)** To achieve accuracy, we must address the CPU contention delays ($N \times t$) in basic colocation (Figure 6.1b). We propose emulating CPU-intensive processing with *processing illusion* (PIL), which replaces an actual processing with `sleep()`. With PIL, an expensive function will sleep and wake up in accurate time with the correct output. As illustrated in Figure 6.1c, if some computations can be emulated with `sleep()` and the output data is automatically generated given the input data, then the resulting time is more accurate ($t+e$) to the one in real-scale testing.

PIL extends the intuition behind data-space emulation [121], where the insight is: "how data is processed is not affected by the content of the data being written, but only by its size." For PIL, our insight is that *"the key to computation is not the intermediate results, but rather the execution time and eventual output."* In other words, what matters is the global cascading implication of the long execution time of the individual nodes.

• **Finding PIL-safe and offending functions:** One key question PIL method raises is: which functions can be safely replaced with `sleep()` *without* changing the whole processing semantic? We name them "PIL-safe functions/code blocks." We set a rule that a PIL-safe function must have a memoizable output (*i.e.*, a deterministic output on a given input) and not have any side effects

such as disk I/Os, network messages, and blocking mechanisms such as locks. Many functions satisfy the rule above, but not all PIL-safe functions should "take the PIL"; that is, they might not be the "offending" functions that lead to scalability bugs. Thus, we raise another key question: which functions are offending?

We learned that many offending functions contain loops that are cluster-size dependent (*e.g.*, a `for`-loop that iterates a cluster-ring data structure). Some of the loops can also be a nested loop. Finding such code blocks are unfortunately not straightforward. Scale-dependent loops can span across multiple functions; in CA-6127, $O(N^3)$ loops span 1000+ LOC across 9 functions. Moreover, they can be inside some `if-else` branches reachable only from a certain path/workload; in CA-6127, the last $O(N^2)$ loop is only exercised if the cluster bootstraps from scratch. All of these suggest that finding PIL-safe and offending functions require an advanced program analysis (which we discuss later). Such a tool will guide the developers to decide which paths/protocols to test, to uncover potential scalability bugs.

• **Memoizing PIL-replaced functions:** PIL-safe and offending functions will become "PIL-replaced functions" where their actual processing will be skipped during replays with `sleep(t)`. Thus, two more questions to address are: how to produce the output if the actual computation is skipped and how to predict the actual compute time (`t`) accurately?

The answer to the first question is *pre-memoization*. That is, given a PIL-replaced code block, we need to first execute the code block and record the input/output around it. The only way to do this on a single machine is to run the protocol with basic colocation, which will consume some time due to the CPU contention delays. However, this will only be a *one-time* overhead, while the fast PIL-infused replay stage can be repeated numerous times without contention.

It is challenging to pre-memoize PIL-replaced functions with an offline input-sampling method without running the protocol at least once. The reason is that, in the context of large-scale, decentralized, non-deterministic distributed systems, covering all possible input/output pairs may require an "infinite" time and storage space. In other words, input/output pairs depend on the precise order

of message arrivals, which can be random. In a ring rebalancing algorithm for example, with N nodes and P partitions/node, there are $(N^{NP})^2$ input/output pairs given all possible orderings. Thus, to cap the state space, the pre-memoization stage also records message ordering, which will be deterministically enforced during PIL-infused replay. With this "order determinism," we do not have to record all possible input/output pairs. We simply record pairs that are observed in one particular run of the protocol test.

The answer to the second question (predicting `t`) is *in-situ time recording*; in addition to storing input/output pairs we also store input/duration pairs (Figure 6.2c). It is almost impossible to predict compute time with a prediction/static-analysis approach. As mentioned above, nested loops can span across multiple functions with many `if-else` conditions. In a Cassandra bug, the duration of an offending code block can range from 0.001 to 4 seconds depending on multi-dimensional inputs. One might also wonder whether time recording is enough to hint the developers of the potential scalability bugs (*e.g.*, 4 seconds of compute should raise a red flag). As mentioned earlier, every implementation is unique (Section 5.2); for example, in CA-5456, if the lock is fine-grained, the long compute will not cause cascading impacts. Furthermore, patches of scalability bugs do not always remove the expensive computation. Put simply, scalability bugs are not merely about the expensive functions, but rather their global implications.

6.3 Evaluation

We now evaluate our SCK integration to Cassandra. Our evaluation answers the following questions. Section 6.3.1: How accurate is SCK compared to real deployments? Section 6.3.2: Can SCK find old scalability bugs? Section 6.3.3: Can SCK reveal new bugs? Section 6.3.4: Does our evaluation compare well with other work?

We use the Nome cluster; each machine has 16-core AMD Opteron(tm) 8454 processors with 32-GB DRAM [23]. To measure SCK accuracy (Section 6.3.1), we compare it with real deploy-

ments of 32, 64, 128, 256, and 512 nodes, deployed on at most 128 Nome machines.[1]

6.3.1 Accuracy

Next, we provide a detailed accuracy evaluation of SCK. Due to space constraints, this section only focuses on one bug (CA-6127 [13]) while the next section briefly discuss other bugs we reproduced.

Figure 6.3a-d presents the internal metrics within Cassandra failure detection protocol that we measured for *every pair* of nodes. That is, the algorithm runs on every node A for every peer B. Figure 6.4a-d compare in detail the accuracy of SCK compared to real deployments. For example, x=512 implies the comparison of 512-node colocation in SCK versus a real deployment of 512 nodes. Note that for CA-6127, we only need time profiling with offline sampling and no pre-memoized data.

Figure 6.4a shows the total number of flaps (alive-to-dead transitions) observed in the whole cluster during bootstrapping. As shown, SCK closely mimics real deployment scenarios. Most importantly here, a significant *#flaps* does not appear until 256-node deployment, hence mini-cluster extrapolation techniques will not work (Section 6.1). Figure 6.3a defines that *#flaps* depends on Φ [68]. Every node A maintains a Φ value for a peer node B (a total of $N \times (N-1)$ variables to monitor). If $\Phi > 8$ for B, A will declare B dead (a flap).

Figure 6.4b shows the maximum Φ values observed for every peer node. For example, for the 512-node setup, the whisker plots show the distribution of the maximum Φ values observed for each of the 512 nodes. As shown, the larger the cluster, more Φ values exceeds the threshold value of 8, hence the flapping. Figure 6.3b points that Φ depends on the average inter-arrival time of when new gossips about B arrives at A ($T_{avgGossip}$) and the time since A heard the last gossip about B ($T_{lastGossip}$); the "last gossip" is the last version number received (Section 5.1.1). The

1. The Nome cluster only has 256 nodes and near the publication deadline, we need to fair-share the cluster with other groups. Our target protocols only make at most 2 busy cores per node, which justifies why we run 8 nodes per one 16-core machine for the real deployment.

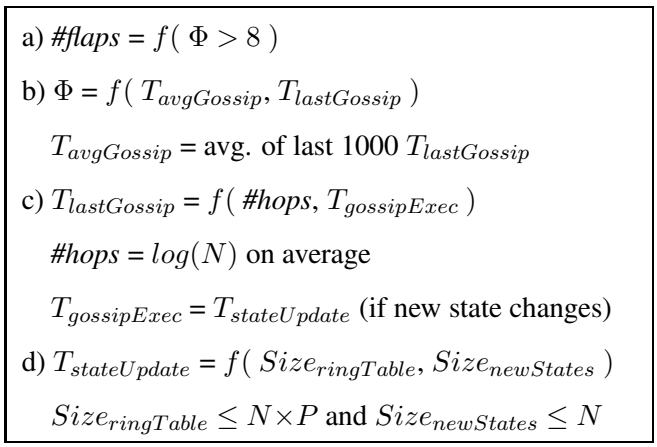

Figure 6.3: **Cassandra internal metrics (Section 6.3.1).** *Above are the metrics we measured within the Cassandra bootstrap protocol for measuring* SCK *accuracy (Figure 6.4). "f" represents "a function of" (i.e., an arbitrary function).*

point is that $T_{lastGossip}$ should not be much higher than $T_{avgGossip}$.

Figure 6.4c shows the whisker plots of gossip inter-arrival times ($T_{lastGossip}$) that we collected for every A-B pair. For example, for the 512-node setup, the whisker plots represent the distribution of around 41 million gossip inter-arrival times; this large number is because a message contains gossips of many peer nodes. The figure shows that in larger clusters, new gossips do not arrive as fast as in smaller clusters, especially at high percentiles. Figure 6.3c shows that $T_{lastGossip}$ depends on how far B's new gossips propagate through other nodes to A (*#hops*) and the gossip processing time in each hop ($T_{gossipExec}$). The *#hops* is stable at $log(N)$ on average in SCK and real deployment (not shown). The latter ($T_{gossipExec}$) is essentially state-update processing time ($T_{stateUpdate}$) whenever there are state changes, which is the culprit.

Figure 6.4d (*in log scale*) shows the whisker plots of state-update processing time ($T_{stateUpdate}$); in the 512-node setup, we measured around 25,000 state-update invocations. The figure shows that at high percentiles, $T_{stateUpdate}$ is scale-dependent. As explained in Figure 6.3d (and Section 5.1.1), $T_{stateUpdate}$ complicatedly depends on a scale-dependent 2-dimensional input ($Size_{ringTable}$ and $Size_{newStates}$); a node's $Size_{ringTable}$ depends on how many nodes it knows, including the partition arrangement ($\leq N \times P$) and $Size_{newStates}$ ($\leq N$) increases as cluster size increases. Note

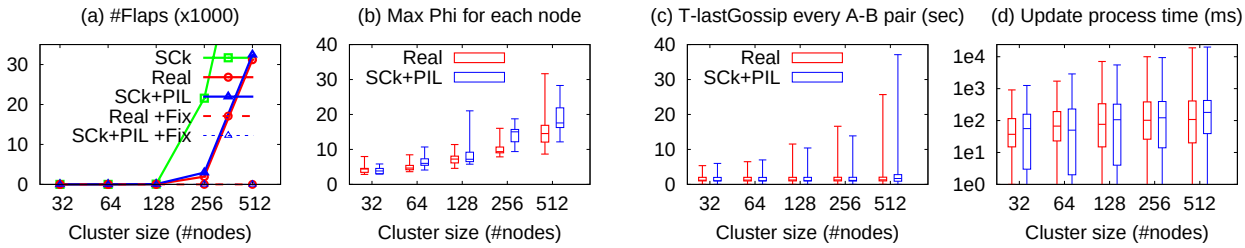

Figure 6.4: **Accuracy in reproducing CA-6127 (Section 6.3.1).** *The figures represent the metrics presented in Figure 6.3, measured in real deployment ("Real") and* SCk *with different cluster sizes (32, 64, 128, 256, and 512). Figure title represents the y-axis.*

that the $T_{stateUpdate}$ in SCk comes from the sampling-based time profiling, which is relatively accurate as the figure shows.

We conclude that SCk mimics similar behaviors as in real deployments and is accurate for reproducing scalability bugs. As an additional note, we have applied the bug patch in both SCk and real deployment modes; Figure 6.4a shows *#flaps* is always zero in both modes.

6.3.2 Bugs Reproduced

Table 6.1 lists all the 7bugs[2] we have reproduced (4 Cassandra bugs). We chose these 7bugs (among the 12bugs we studied) because the reports contain more detailed descriptions about the bugs, the affected protocols, the affected code version numbers, configuration setups, and the patches. Table 6.1 also shows the number of nodes needed for the bug symptoms to surface and the quantifiable metrics of the symptoms. Our first target system was Cassandra, hence the more bugs reproduced compared to Riak and Voldemort; the latter two were added for stronger proof of concept. Figure 6.5 shows the accuracy of SCk in reproducing the 6 bugs using the metrics shown in Table 6.1. The first bug, CA-6127, has been described in Section 5.1.1 and Section 6.3.1. We now briefly discuss the other five bugs (shown in Figure 6.5) and then make several important remarks.

Figure 6.5a: In CA-3831[10], when a node D is decommissioned from a large cluster, all other

2. Given the limited time and number of students.

Bug#	Surface	Protocol	Metric
CA-6127 [13]	$N \geq 256$	Bootstrap	*#flaps*
CA-3831[10]	≥ 256	Decommission	*#flaps*
CA-3881[11]	≥ 64	Add new nodes	*#flaps*
CA-5456[12]	≥ 256	Add new nodes	*#flaps*

Table 6.1: **Reproduced bugs (Section 6.3.2).** *"Surface" implies the number of nodes needed for the bug symptom to surface. "c" stands for Cassandra, "r" for Riak, and "v" for Voldemort.*

nodes must own D's key-partitions. This scale-dependent "pending keyrange calculation" is CPU intensive, causing cluster-wide flapping, significantly observable in 256+ nodes. The developers fixed it by caching outputs of slow methods.

Figure 6.5b: CA-3881[11] is similar to the previous bug (CA-3831), but the fix was not efficient enough because in this new bug, the concept of "virtual nodes" (multiple key-partitions per node) was added to Cassandra. The calculation is now scale-dependent to $N \times P$ and becomes very CPU intensive. This causes massive flapping during scaling out; the bug surfaced in 64+ nodes (when 32+ new nodes are added to existing 32+ nodes). The bug was fixed with a complete redesign of the pending keyrange calculation.

Figure 6.5c: Interestingly, CA-5456[12] is a bug in the *same* protocol as in the previous bug. We wondered why the previous fix does not work here. We found that pending range calculation is now multi-threaded; different range calculations can happen concurrently. This new design however introduces a new coarse-grained lock that creates a new problem; it can block gossip processing for a long time, thus introduce flapping (in 256+ nodes). The fix changed the lock management. The figure also shows that, for this workload, offline time profiling is not fully accurate as the bug is order sensitive at large scale. We are now testing it with order determinism.

We make several remarks from this experience. First, if SCK had existed in the first place, it might have *prevented* the Cassandra bugs; they all involve the same protocols (gossip, rebalance, and failure detector) and create the same symptom (high *#flaps*). These bugs highlight that code evolution can introduce new bugs in the same protocols. In this context, SCK is highly useful. Second, reproducing scalability bugs is relatively *easy* as we achieve a high colocation factor.

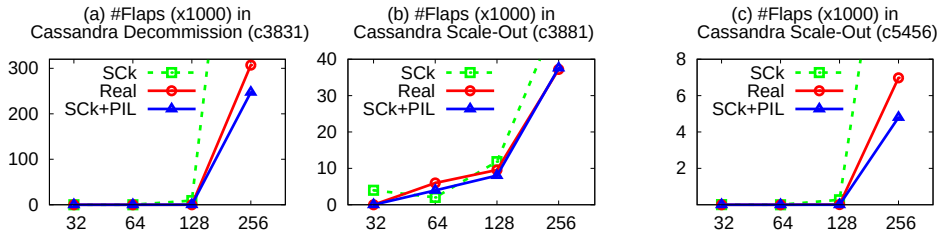

Figure 6.5: **Accuracy in reproducing other bugs (Section 6.3.2).** *The figures represent the bugs described in Table 6.1. The title represents the y-axis. We cap the y-axis to show the scale at which the bug symptoms start to appear.*

Unlike non-deterministic bugs which require complex timing reordering to reproduce [63, 85], symptoms of scalability bugs are "deterministically scale-dependent." Third, different systems of the same type (*e.g.*, key-value store) implement similar protocols. The *generality* of SCK methods in scale-checking the protocols above can be useful to many other distributed key-value stores.

6.3.3 New Bugs

We also scale-checked the latest stable versions of Cassandra (v2.2.5), Riak (v2.1.3), and Voldemort (v1.10.21). In Cassandra, SCK shows that cluster-wide flapping resurfaces again but only observable in 512-node deployment (*e.g.*, decommissioning only one node caused almost 100,000 flaps). We submitted the bug few months back and it is still unresolved (the fix might require new design). Meanwhile, the developers suggested us to add/remove node one at a time with 2-minute separation, which means scaling-out/down 100 nodes will take over 3 hours; instant elasticity is not achievable. We then found out that Cassandra developers just recently started a new initiative and opened a new "umbrella" ticket (July 2016) for designing "Gossip 2.0" [20], supposed to scale to 1000+ nodes; the conversation just begun [21].

6.3.4 Evaluation Scope (vs. Other Work)

1. **Colocation factor:** DieCast is primarily evaluated with 10 nodes/machine. Exalt runs up to 100 nodes per 16-core machine SCK can achieve up to 512colocation factor. However, in our view, Exalt and SCK *complement each other* as they target different types of bugs (data-

vs. compute-intensive).

2. **#Target systems:** Exalt is integrated to two master-slave systems (HBase and HDFS), DieCast to 3 systems (BitTorrent, RUBis, and ISaaC), and SCK to three P2P key-value stores.

3. **#Target protocols:** Exalt and DieCast test write protocols and SCK tests 5control-path protocols.

4. **#Bugs:** DieCast did not reproduce any bugs; it mainly evaluates throughput/latency accuracy. Exalt discussed 6 bugs in total (5 out of 6 are bugs in the Namenode, the non-emulated node; Section 6.1). SCK reproduced 6 bugs in emulated P2P nodes.

5. **Types of bugs:** While most work focus on data-plane bugs (Section **??**, Section 6.1), SCK focuses on control-plane protocols which are mainly about cluster stability (no flapping, eventually balanced, etc.).

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Summary

In this dissertation, we aim to unearth hidden bugs in cloud-scale distributed systems that weaken reliability and scalability of the systems. For reliability bugs, we focus in distributed concurrency (DC) bugs. For scalability bugs, we focus in CPU-intensive protocol bug.

The key idea for unearthing DC bugs is advancing model checking techniques for dmcks to tackle state-space explosion. Because existing dmcks treat every target system as a complete *black box*, and therefore perform unnecessary reorderings of distributed events that would lead to the same explored states (*i.e.*, redundant executions). To tackle the problem, we introduced Semantic-Aware Model Checking (SAMC), a novel white-box model checking approach that takes *semantic knowledge* of how distributed events (specifically, messages, crashes, and reboots) are processed by the target system and incorporates that information in reduction policies. The policies are based on sound reduction techniques such as DPOR and symmetry.

And for scalability bugs, we believe from the study that we need to scale check distributed system implementation at real scale, not via simulation nor extraploation. Our proposed solution is to colocate as many nodes as possible (*e.g.*, hundreds) on one machine without sacrificing accuracy by *emulate* hardware resources such that the individual nodes behave as if they run on independent machines. We propose four emulation concepts to mitigate hardware contention problems. We are exploring these concepts to build scale check methodology for cloud-scale distributed systems.

7.2 Future Work

In this work we have built a number of reduction policies that exploit simple message processing patterns, failure recovery patterns (*e.g.*, discard pattern, local-only update pattern), and reboot synchronization patterns. There are two follow-up questions to ask. First, what and how much

semantic knowledge can advance dmck reduction policies? Second, How to extract that semantic knowledge automatically from the target system? We will address these two questions in this chapter. In addition, we will discuss about finding bugs that caused by both local and distributed interleaving.

7.2.1 Broad SAMC

Ideally we want to expose as much as semantic knowledge that can help dmck unearth deep bugs without sacrificing soundness. With current SAMC we believe we have only addressed the surface of the power of semantic-awareness for dmck. We will continue digging the semantic-awareness to push SAMC to become broader or we call it "Broad SAMC", a dmck that incorporates a much broader semantic knowledge to find deep bugs in several distributed protocols.

To begin with, We will start by applying SAMC in model checking various kind of distributed systems. By learning how messages and failures are processed in a diverse set of distributed protocols, we will come up with a much broader generic semantic knowledge that can be incorporated into dmck.

Next, with many type of fault-injection mechanisms, we will exercise more complex failures (*e.g.*, timeouts, disk failures). Although we have exercised crashes and reboots in current SAMC, and gained some understanding of crash recovery and reboot semantics which helped us create policies that reduce unnecessary crashes and reboots and their re-orderings. But, failures are diverse and bugs can manifest from a variety of failures such as timeout events (MR-3596, MR-4833). The current SAMC cannot find these bugs yet, so we need to exercise timeouts and understand the interaction between them and other distributed events and incorporate the knowledge to dmck reduction policies. Overall, by exercising diverse failures (*e.g.*, timeouts, disk failures) and building reduction policies that covers more type of failures, we will unearth more deep bugs.

7.2.2 *Auto SAMC*

To answer the second question, how to extract semantic knowledge automatically from the target system, we have considered our current approach to extract knowledge. So far SAMC requires developers to manually extract semantic knowledge for dmck and write the corresponding reduction policies manually. This manual step is based on high-level human understanding of the code, which can be error-prone. The developers can also introduce human errors when writing reduction policies and breaks soundness. Thus, when the current SAMC does not find bugs, it does not imply systems are bug-free; some bugs might be accidentally missed my wrong policies. Currently, there is no way to verify that the semantics used for pruning out executions is correct.

To address these limitations, we propose "Auto SAMC", a dmck that automatically extracts useful semantic knowledge to build reduction policies. To achieve this goal, we will create an advanced source code analysis tool that combines symbolic execution and SAMC. Symbolic execution generates constraints at condition statements that define predicate values leading to true and false conditions. The collection of condition constraints leading to a specific path in the code is called a path constraint. Others have used symbolic execution to target system problems [**?**], but Auto SAMC is different. For instance, we want to build path constraints that lead to state updates but whose states stay the same regardless of the reordering (*e.g.*, discard, increment, commutative patterns) or path constraints that lead to state-update independence. Each path constraint can be a function of the local state of each node and the messages the node receives. Generating such a path constraint will require a symbolic execution tool with additional new supports.

7.2.3 *LocDist SAMC*

Some bugs in distributed systems are caused by interleavings of local and distributed events such as MR-3721; this kind of bugs exacerbates the state-space explosion problem. This is an unsolved problem. We do not see any work touch these two types of events together; we know only one work that concerns local interleavings [63]; here, the authors use a technique called dynamic-interface

reduction (DIR) which essentially caches and replays global re-ordering inputs and outputs without re-running them when the local interleavings produce the same global messages.

To address the problem of local and distributed interleavings, We will build "LocDist SAMC", a dmck that incorporates both interleavings and use both local and distributed processing semantic awareness to reduce the number of re-orderings. We will build LocDist SAMC on top of Broad and Auto SAMC.

As presented in this thesis, so far we only focus on semantics that reduce distributed interleavings. In LocDist SAMC, we will leverage semantics related to local interleavings. For example, in MR-3721, the bug appears when a reducers fetcher thread runs faster than its merger thread that is against a design assumption (fetcher usually runs slower due to network fetches, while merger is fast due to in-memory operation). Here, it is unnecessary to interleave every instance of data processing between the fetcher and merger; interleaving the termination points of fetcher and merger might suffice. As another example, thread interleavings in adding jobs to a queue might not matter. Overall, understanding the semantics of local processing can help prune down unnecessary exectuions.

REFERENCES

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

[2] Adobe Creative Cloud Adoption Grows to 9 Million Paid Members. http://prodesigntools.com/creative-cloud-one-million-paid-members.html.

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

[4] Apache Cassandra. https://en.wikipedia.org/wiki/Apache_Cassandra.

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

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

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

[8] AWS Customer Success. https://aws.amazon.com/solutions/case-studies/.

[9] Big data storage: Hadoop storage basics. http://www.computerweekly.com/feature/Big-data-storage-Hadoop-storage-basics.

[10] BUG: CASSANDRA-3831: scaling to large clusters in GossipStage impossible due to calculatePendingRanges. https://issues.apache.org/jira/browse/CASSANDRA-3831.

[11] BUG: CASSANDRA-3881: reduce computational complexity of processing topology changes. https://issues.apache.org/jira/browse/CASSANDRA-3881.

[12] BUG: CASSANDRA-5456: Large number of bootstrapping nodes cause gossip to stop working. https://issues.apache.org/jira/browse/CASSANDRA-5456.

[13] BUG: CASSANDRA-6127: vnodes don't scale to hundreds of nodes. https://issues.apache.org/jira/browse/CASSANDRA-6127.

[14] Database Scalability: Vertical Scaling vs Horizontal Scaling. http://www.vcloudnews.com/database-scalability-vertical-scaling-vs-horizontal-scaling/.

[15] Dropbox. https://www.dropbox.com/.

[16] Gmail. https://www.google.com/gmail/about.

[17] Gmail Now Has More Than 1B Monthly Active Users. https://techcrunch.com/2016/02/01/gmail-now-has-more-than-1b-monthly-active-users/.

[18] Google Drive. https://www.google.com/intl/en/drive/.

[19] Google Drive claims one million paying customers, er, organizations. http://fortune.com/2015/09/21/google-drive-1m-paid-users/.

[20] Gossip 2.0. https://issues.apache.org/jira/browse/CASSANDRA-12345.

[21] Gossip 2.0. http://mail-archives.apache.org/mod_mbox/cassandra-dev/201609.mbox/%3CCAHjqPuJMkfZwp9DDX45PNBNhkoGXsPW4TFT6Zxv%2BTTz_Pg3Y%2Bg%40mail.gmail.com%3E.

[22] iCloud. https://www.icloud.com/.

[23] NMC PRObE Nome Nodes. https://www.nmc-probe.org/wiki/Nome:Nodes.

[24] Number of registered Dropbox users from April 2011 to March 2016 (in millions). https://www.statista.com/statistics/261820/number-of-registered-dropbox-users/.

[25] Siri. https://www.apple.com/ios/siri/.

[26] Taking astronomy to the cloud. https://www.inthefieldstories.net/taking-astronomy-to-the-cloud/.

[27] The Top 20 Valuable Facebook Statistics - Updated May 2017. https://zephoria.com/top-15-valuable-facebook-statistics/.

[28] Twenty-One Experts Define Cloud Computing. http://cloudcomputing.sys-con.com/node/612375/print.

[29] What is cloud computing? https://www.ibm.com/cloud-computing/learn-more/what-is-cloud-computing/.

[30] What Is Cloud Computing? http://www.pcmag.com/article2/0,2817,2372163,00.asp.

[31] Running Netflix on Cassandra in the Cloud. https://www.youtube.com/watch?v=97VBdgIgcCU, 2013.

[32] Why the world's largest Hadoop installation may soon become the norm. http://www.techrepublic.com/article/why-the-worlds-largest-hadoop-installation-may-soon-become-the-norm/, 2014.

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

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

[35] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[36] Rajkumar Buyya, Srikumar Venugopal Chee Shin Yeo and, James Broberg, and Ivona Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing As the 5th Utility. In *Future Generation Computer Systems*, 2009.

[37] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[38] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[39] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[40] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[41] 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 *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[42] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry reductions in model checking. In *10th International Conference on Computer Aided Verification (CAV)*, 1998.

[43] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

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

[45] 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 *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2007.

[46] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe Asynchronous Event-Driven Programming. In *Proceedings of the*

ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation (PLDI), 2013.

[47] 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 *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[48] E. Allen Emerson, Somesh Jha, and Doron Peled. Combining Partial Order and Symmetry Reductions. In *The 3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1997.

[49] 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.

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

[51] Loek Essers. Cloud Failures Cost More Than $70 Million Since 2007, Researchers Estimate. http://www.pcworld.com, 2012.

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

[53] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

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

[55] Ian Foster, Young Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *2008 Grid Computing Environments Workshop*, 2008.

[56] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[57] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay Debugging for Distributed Applications. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2006.

[58] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[59] Patrice Godefroid. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. volume 1032, 1996.

[60] Rachid Guerraoui and Maysam Yabandeh. Model Checking a Networked System Without the Network. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[61] 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 *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[62] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[63] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[64] Diwaker Gupta, Kashi Venkatesh Vishwanath, and Amin Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[65] Diwaker Gupta, Kenmeth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[66] 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 *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[67] 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 *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[68] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The Phi Accrual Failure Detector. In *The 23rd Symposium on Reliable Distributed Systems (SRDS)*, 2004.

[69] Brian Hayes. Cloud Computing. In *Communications of the ACM*, 2008.

[70] 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 *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[71] 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.

[72] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. DDOS: Taming Nondeterminism in Distributed Systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[73] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[74] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. Radbench: A concurrency bug benchmark suite. In *The 3rd USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*, 2011.

[75] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI)*, 2012.

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

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

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

[79] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[80] Xiaoen Ju, Livio Soares, Kang G. Shin, Kyung Dong Ryu, and Dilma Da Silva. On Fault Resilience of OpenStack. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

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

[82] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[83] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, and Feng Qin. Debugging High-Performance Computing Applications at Massive Scales. *Communications of the ACM (CACM)*, 58(9), September 2015.

[84] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2009.

[85] 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 *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[86] 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 *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013.

[87] Xin Li, Michael C. Huang, and Kai Shen. An Empirical Study of Memory Hardware Errors in A Server Farm. In *The 3rd Workshop on Hot Topics in System Dependability (HotDep)*, 2007.

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

[89] Thomas A. Limoncelli and Doug Hughe. LISA '11 Theme – DevOps: New Challenges, Proven Values. *USENIX ;login: Magazine*, 36(4), August 2011.

[90] David Linthicum. Calculating the true cost of cloud outages. http://www.infoworld.com, 2013.

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

[92] 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 *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[93] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[94] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[95] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

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

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

[98] Jake Luo, Min Wu, Deepika Gopukumar, , and Yiqing Zhao. Big Data Application in Biomedical Research and Health Care: A Literature Review. In *Biomedical Informatics Insights*, 2016.

[99] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computing - The business perspective. In *Decision Support Systems*, 2011.

[100] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB)*, 2010.

[101] Maged Michael, Jose E. Moreira, Doron Shiloach, and Robert W. Wisniewski. Scale-up x Scale-out: A Case Study using Nutch/Lucene. In *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007.

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

[103] John Ousterhout. Is Scale Your Enemy, Or Is Scale Your Friend?: Technical Perspective. *Communications of the ACM (CACM)*, 54(7), July 2011.

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

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

[106] 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.

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

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

[109] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[110] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An Empirical Study of Reported Bugs in Server Software with Implications for Automated Bug Diagnosis. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, 2010.

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

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

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

[114] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV)*, 2010.

[115] Jiri Simsa, Randy Bryant, Garth A. Gibson, and Jason Hickey. Scalable Dynamic Partial Order Reduction. In *The 3rd International Conference on Runtime Verification (RV)*, 2012.

[116] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 2010.

[117] Vilas Sridharan and Dean Liberty. A Study of DRAM Failures in the Field. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[118] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[119] 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.

[120] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[121] Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

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

[123] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Tom Anderson. Verdi: A framework for formally verifying distributed system implementations. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[124] 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 *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[125] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[126] 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 *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[127] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[128] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT international symposium on Foundations of software engineering (FSE)*, 2011.

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

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

[131] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[132] 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.