Mitigating Distributed Configuration Errors in Cloud Systems

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Sixiang Ma, M.E.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2022

Dissertation Committee:

Dr. Yang Wang, Advisor

Dr. Michael D. Bond

Dr. Feng Qin

Dr. Kannan Srinivasan

Dr. Xiaoyi Lu

Abstract

While many techniques have been proposed to find software configuration errors in software systems, most of them focus on finding misconfiguration occurring on a single node. Unfortunately, the nature of distributed systems brings up a more complex problem: some failures may only occur when a system is configured inappropriately on multiple nodes, whereas the configuration of each node is considered correct individually. To distinguish these configuration errors from local configuration errors which have been widely studied, we call these errors as distributed configuration errors. In this dissertation, we combat distributed configuration errors in two ways: 1) we re-design the system to reduce the chance that the administrator may introduce an inappropriate distributed configuration; 2) we use the traditional software testing approach to test what distributed configurations are unsafe.

In the first direction, we focus on timeout, an important parameter that is hard to configure right. We propose SafeTimer, a mechanism to enhance existing timeout failure detection protocols to tolerate long delays in the OS and the application: at the heartbeat receiver, SafeTimer checks whether there are any pending heartbeats before reporting a failure; at the heartbeat sender, SafeTimer blocks the sender if it cannot send out heartbeats in time. As a result, as long as networking delays are bounded, SafeTimer can guarantee the correctness of failure detection. We applied SafeTimer to HDFS and Ceph with little modification, and found the performance overhead is small.

In the second direction, we propose ZebraConf, a testing framework that reuses existing unit tests and integration tests to test whether a parameter can be configured in a heterogeneous manner. To address the challenge of assigning different configurations to different nodes in unit tests, ZebraConf incorporates several heuristics to accurately map configuration objects to nodes. To reduce the massive test number, ZebraConf profiles unit test suites to only generate effective tests and groups multiple tests into a single one. We applied ZebraConf to five cloud systems and found 47 heterogeneous-unsafe configuration parameters.

Dedicated to my parents, my wife, and my dogs.

Acknowledgments

I can never reach this stage without the help of many people in my life. I would like to take this opportunity to thank them for their support.

First, I would like to thank my advisor, Dr. Yang Wang, for his invaluable advice and consistent support during the course of my PhD study. His insight and guidance encourage me in all the time of my academic research and daily life.

Second, I would like to thank the other members of my dissertation committee – Dr. Michael D. Bond, Dr. Feng Qin and Dr. Kannan Srinivasan, and Dr. Xiaoyi Lu. This dissertation could not be completed without their thoughtful suggestions and comments.

Finally, I want to thank my wife, Yige Zhou, who has supported me throughout this journey in the past six years with her huge patience and understanding.

Vita

2008-2012 ..... B.E., Software Engineering, Xiamen University, China

2012-2015 ..... M.E., Software Engineering, Shanghai Jiao Tong University, China

2015-Present ..... Ph.D. Computer Science and Engineering, The Ohio State University, U.S.A

Publications

Research Publications

Sixiang Ma, Fang Zhou, Michael D Bond, Yang Wang, "Finding Heterogeneous-Unsafe Configuration Parameters in Cloud Systems". *The 16th European Conference on Computer Systems (EuroSys '21)*, May 2021.

Fang Zhou, Yifan Gan, **Sixiang Ma**, Yang Wang, "Generic Off-CPU Analysis to Identify Critical Waiting Events". *The 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, October 2018.

Sixiang Ma, Yang Wang, "Accurate Timeout Detection Despite Arbitrary Processing Delays". *2018 USENIX Annual Technical Conference (USENIX ATC '18)*, July 2018.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

Page

List of Tables

List of Figures

Figure **Page**

Chapter 1: Introduction

Configuration provides an important way for users to control system behaviors and this mechanism has been used by almost all the systems. When configured inappropriately, errors can happen in systems and cause serious problems. To find misconfiguration in software systems, many techniques have been proposed [3, 50, 79, 92, 93, 96, 98].

While these techniques are helpful to detect and troubleshoot configuration errors on a single node, there are some scenarios of misconfiguration particular to distributed systems that these techniques can have little help.

First, some features in distributed systems are essentially difficult to be configured appropriately. In this scenario, I focus on the configuration of heartbeat timeout, which is widely used in many distributed systems [1, 2, 42, 56, 70, 80, 90, 94] for failure detection. Accurate failure detection is critical to distributed systems, because false failure detection can hurt system correctness, for example, leading to the classic split-brain problem [29] in primary-backup systems. However, configuring heartbeat timeout is difficult, since people need to make tradeoff between correctness and availability: on one hand, using small timeout values can achieve fast failure detection but is prone to false failure detection; on the other hand, using large timeout values can achieve relatively higher accuracy of failure detection but unavoidably makes failure detection slow, hurting system availability. What's worse, even when system administrators intentionally choose to use large timeout values for system

correctness, many reports [28, 39] show that some long-delay events that rarely happen in systems can still fail timeout failure detection, forcing system administrators to further enlarge timeout values to tolerate these events.

Second, with the increasing prevalence of heterogeneous hardware [4,21,31,49,59,66,67, 97,100] and the increasing need for online reconfiguration [7,23,36,40,43,57,68,71,73,81], there is increasing demand for heterogeneous configurations. Heterogeneous configuration, however, may cause the system to fail if not used properly. For example, if one node is configured to encrypt its communication channel while the other node does not decrypt the messages, then unsurprisingly the communication will fail. This type of errors is different from the configuration errors caused by invalid configuration values: in our case, both configuration values (i.e., using and not using encryption) are valid; the problem is caused by two nodes with different configurations communicating with each other.

From these two scenarios, we can see the correctness of these configurations cannot be achieved by configuring each node individually; instead, *the correct configuration is determined by multiple nodes in a distributed system as a whole*. For the first scenario, the accuracy of heartbeat timeout is determined by the heartbeat sender, the heartbeat receiver, and the communication channel between them. For the second scenario, the safety of heterogeneous configuration is determined by the nodes communicating with each other. To distinguish this problem from previously widely studied local configuration errors, in this dissertation, I call configuration errors that are contributed by multiple nodes in a distributed system as *distributed configuration errors*. Because of the nature of these problems, previous techniques of detecting local configuration errors are ineffective to detect this type of configuration errors.

This dissertation tries to mitigate the problem with two different approaches.

First, to reduce the chance that distributed configuration errors can happen in heartbeat failure detection, I present a mechanism called *SafeTimer*, which enhances existing heartbeat failure detection protocols to tolerate long processing delays in the OS and the application layer. At the heartbeat receiver, SafeTimer checks whether there are any pending heartbeats before reporting a failure; at the heartbeat sender, SafeTimer blocks the sender if it cannot send out heartbeats in time. This property allows existing protocols to relax their timing assumptions and use a shorter timeout interval for faster failure detection. As a result, when configuring heartbeat timeout, system administrators only need to make timing assumptions about networking layer. Our evaluation shows that the overhead of SafeTimer is small and applying SafeTimer to existing systems is easy.

Second, to find the heterogeneous configurations that can cause distributed configuration errors, I build a testing framework called *ZebraConf*, which uses the traditional software testing approach to test heterogeneous configurations. To test which configuration parameters are unsafe when configured in a heterogeneous manner, ZebraConf reuses existing unit tests and integration tests but runs them with heterogeneous configurations. To address the challenge that unit tests often share the configuration across different nodes, we incorporate several heuristics to accurately map configuration objects to nodes. To address the challenge that there are too many tests to run, we (1) "pre-run" tests to determine effective tests for each configuration parameter and (2) introduce pooled testing to test several parameters together. Our evaluation finds 47 heterogeneous-unsafe configuration parameters in 6 cloud systems. Sys administrators need to be cautious when configuring these parameters to avoid distributed configuration errors.

Chapter 2: Accurate Timeout Detection Despite Processing Delays

To reduce the chance that distributed configuration errors can happen due to the inappropriate configuration of heartbeat timeout, this chapter presents SafeTimer, a mechanism to enhance existing timeout detection protocols to prevent false failure reports caused by long delays in the OS and the application. With the help of SafeTimer, existing protocols can relax their timing assumptions and thus system administrators do not need to accurately estimate the maximum delays in the OS and the application layer for system correctness.

Timeout is widely used in distributed systems to detect failures [2, 10, 19, 35, 51, 83]: a node periodically sends a heartbeat packet to others and if the receiver does not receive the heartbeat in time, it may report a failure and may take actions to recover the failure.

Although this idea is simple, delays of packet transfer create a problem: if a receiver misses a heartbeat, is it because the sender has not sent the heartbeat, which indicates a failure, or is it because the heartbeat is delayed somewhere, which should not indicate a failure?

To address this problem, existing systems use one of the following approaches: the first is to prevent false failure reports by setting an appropriate timeout interval. However, such setting requires certain timing assumptions about the communication channel [8, 9, 27] and creates a dilemma: on one hand, these assumptions should be conservative enough to tolerate abnormal events that can cause long delays (e.g., congestion), which means the timeout

interval should be long. On the other hand, long timeout interval can hurt system availability, because the system has to wait for a long time before recovering the failure. A recent study shows that inappropriate timeout interval is a major cause of timeout related bugs, leading to various problems like data loss or system hanging [28]. The second approach is to ensure correctness despite false failure reports, using protocols like Paxos [60, 61, 77]. This approach allows short timeout for better availability, but its cost is usually higher.

SafeTimer enhances the first approach to tolerate a subset of those abnormal events, without requiring any timing assumptions. It thus allows existing protocols to relax their timing assumptions to use a shorter timeout interval, without sacrificing the accuracy of timeout detection. It is motivated by two insights.

First, conservative assumptions are only necessary if the communication channel is a blackbox, which cannot provide any additional information other than receiving a packet. If the channel can tell whether a packet is pending or dropped, the receiver can simply check whether there is a pending or dropped heartbeat when missing a heartbeat. This approach can prevent false failure reports without requiring any timing assumptions.

Second, we observe that modeling the whole communication channel as a blackbox is too pessimistic: the routing layer usually does not provide the users with information like packet drops, so it is reasonable to model routing as a blackbox; the OS and the application, however, can provide precise information about its packet processing and thus could be modeled as a whitebox. Furthermore, in today's datacenters, the whitebox part often incurs delays that are comparable to or even larger than those of the blackbox part: on one hand, intra-datacenter networking delays usually range from tens of microseconds to a few milliseconds and can be further reduced to hundreds of nanoseconds with techniques like Infiniband [53]. Improvement in bandwidth and protocols [22, 84] have significantly reduced

the chances of packet drops. On the other hand, a traditional OS can delay processing by several milliseconds because of time sharing or page fault, etc. Such delay can occasionally grow to several seconds for reasons like SSD garbage collection [54] and can grow even higher in abnormal cases.

Because of these two insights—1) the delay of the whitebox part is significant among communication and 2) there exist more effective solutions for the whitebox part—SafeTimer naturally uses a more effective solution for the whitebox part; for the blackbox part, SafeTimer relies on existing protocols and their assumptions.

At the receiver side, SafeTimer guarantees that *as long as the network interface card (NIC) has either delivered or dropped the heartbeat before the deadline, the receiver will not report a failure.* To achieve this property, SafeTimer's receiver module checks whether there are any pending or dropped heartbeats in the system before reporting a failure. Implementing this idea, however, is challenging, because modern OS incorporates a highly concurrent pipeline for fast packet processing. Naive solutions like pausing all its threads requires an intrusive modification to kernel, which is undesirable.

To solve this problem, we propose a non-blocking solution: when the timer expires at t, SafeTimer's receiver module will send a barrier packet to itself. By crafting the barrier packet and configuring the OS properly, SafeTimer ensures that if the receiver module receives the barrier, all heartbeats processed by the NIC before t must have been either delivered to the application or dropped. Therefore, if the receiver module has neither received the heartbeat nor observed any packet drops, it can safely report a failure.

At the sender side, SafeTimer guarantees that *if the sender has not sent out a heartbeat in time, the sender will not be able to send out any new packets.* Such suicide idea is not novel [12, 32], but previous solutions that actively kill or reboot the sender do not work

when considering long processing delays, because the kill or reboot operations may be delayed as well, leaving the sender alive. To solve this problem, SafeTimer incorporates a passive design: SafeTimer's sender module maintains a timestamp to identify till when it is valid for the sender to send new packets. The sender module updates this timestamp when successfully sending a heartbeat and checks this timestamp before sending any packets. By doing so, SafeTimer prevents a sender which fails to send heartbeat in time to affect other nodes in the system.

One can enhance an existing timeout detection protocol by applying SafeTimer at both the sender and the receiver. We can prove that, as long as the existing protocol's assumptions about the blackbox part hold, SafeTimer is accurate (i.e., never report failure for a correct sender) despite arbitrary delays in the whitebox part and is complete (i.e., eventually report failure for a failed sender) when the receiver does not experience slow processing or packet drops for sufficiently long [18]. Such properties indicate that one does not need to make conservative assumptions about the whitebox part, and thus can use a shorter timeout interval to improve availability.

Our evaluation shows that the overhead of SafeTimer is negligible when processing big packets and at most 2.7% when processing small packets; SafeTimer can prevent false failure reports when long processing delays are injected; and applying SafeTimer to HDFS [42] and Ceph [15] is easy.

2.1 Motivation: Long delays in OS and application

SafeTimer allows existing timeout detection protocols to relax their timing assumptions by excluding delays in the OS and the application. To demonstrate the potential benefits of such relaxation, we present a number of abnormal events that can cause long delays.

- **Disk access.** Disk accesses caused by logging heartbeats [51, 83] or page faults can block heartbeat processing. A typical hard drive has an average latency of tens of milliseconds and an SSD usually has a lower average latency. Worst-case latency, however, is much longer: SSD's internal garbage collection can delay an access by more than one second [54]. Our experiment with hard drives shows that when processing frequent random writes, the buffering mechanism in the file system can occasionally introduce a latency of tens of seconds, when it flushes many random writes.
- **Packet processing.** OS kernel can drop packets at different layers when it runs out of buffer space, which can cause extra delay. Furthermore, handling of abnormal packets may cause a significant delay as well. For example, when Linux receives a packet to an unopened port, it will report "port unreachable" to the router using ICMP [52]. In our experiment, a large number of such abnormal packets can delay the processing of heartbeat by more than two seconds.
- **JVM garbage collection.** Garbage collection in a Java Virtual Machine (JVM) can block the execution of the application. Our experiment on a JVM with 32GB of memory shows that when the memory is close to be fully utilized, a single garbage collection can take up to 26 seconds, even when using parallel GC. A recent survey [28] has observed similar problems in ZooKeeper and HBase (HBase-3273 [39]).
- **Applicaton specific delays.** Applications may have specific logics that can cause long delays occasionally. For example, previous works have reported that HDFS DataNode's heartbeat sending thread may be blocked by the task of scanning local data, which could take long [89]. Although newer versions of HDFS have fixed this problem, our investigation shows that similar problems still exist: the heartbeat

sending thread can also be blocked by the task of deleting directories, which can take long as well. A similar problem has been reported in Ceph, in which a heavy rejoin operation can block heartbeat processing [17].

As shown in these examples, some events in the OS and the application can cause delays of tens of seconds, which are comparable to or larger than many systems' default timeout intervals (e.g., 30 seconds in HDFS [48], 5 seconds in ZooKeeper [38], 20 seconds in Ceph [16]). Furthermore, some of these delays may grow longer if a machine has more resource (e.g., more memory for JVM garbage collection).

Existing timeout detection protocols must make their timing assumptions conservative enough to cover all the events mentioned above. For example, to tolerate long garbage collection in ZooKeeper [39], the developers increased their timeout intervals, which will hurt system availability as discussed previously. With the help of SafeTimer, however, they can tolerate these events without requiring any timing assumptions, and thus can use a shorter timeout for faster failure detection.

2.2 Model

The goal of SafeTimer is to enhance existing timeout detection protocols to tolerate long processing delays in the OS and the application. To achieve this goal, SafeTimer makes a few assumptions about the existing protocol: at the receiver side, SafeTimer assumes the receiver defines multiple time intervals and reports a failure if it does not receive any heartbeats during an interval. At the sender side, SafeTimer assumes the application has its own rules to decide when to send heartbeats and whether heartbeats are sent successfully, based on its timing assumptions. Furthermore, SafeTimer assumes these intervals and assumptions are configurable, so that the user can use a shorter timeout interval with the help of SafeTimer.

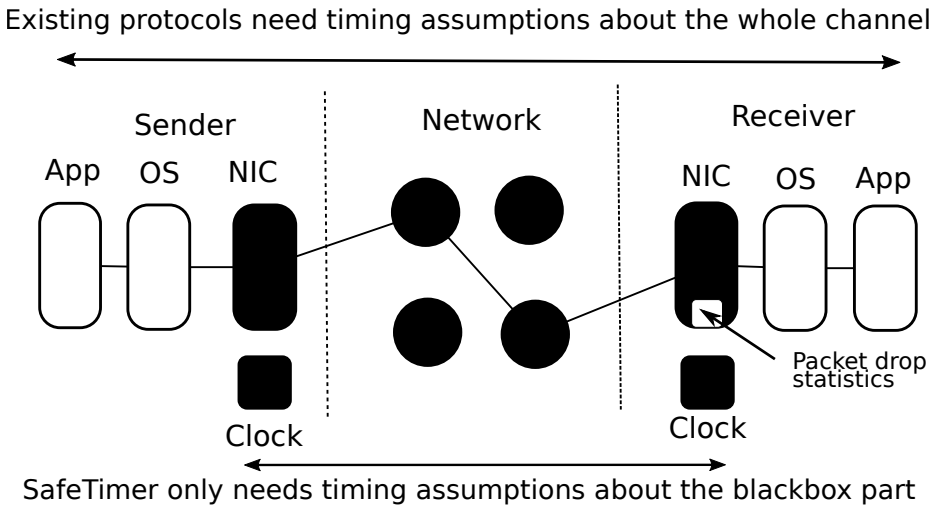

Figure 2.1: System model: SafeTimer can tolerate long delays in the whitebox part without timing assumptions.

SafeTimer enhances existing protocols to tolerate a subset of abnormal events without requiring timing assumptions. Figure 2.1 shows which events SafeTimer can tolerate: the blackbox part includes the network interface cards (NICs) at both sides, the clocks at both sides, and packet routing between two NICs; the whitebox part includes the OS and the application's logic to process packets at both sides. SafeTimer can tolerate long delays in the whitebox part without requiring any timing assumptions. Instead, SafeTimer only assumes that, a node will eventually finish processing a heartbeat and SafeTimer can observe the result (either delivered or dropped). For the blackbox part, SafeTimer relies on existing protocols and their assumptions.

Abnormal events in the whitebox part may affect the processing speed of the blackbox part. SafeTimer assumes such effect can be observed at the boundary: a slow receiver may cause its NIC to drop packets because the receiver's buffer is full and SafeTimer assumes

the NICs can provide packet drop statistics. We find this function is commonly provided by modern NICs.

With the help of SafeTimer, existing timeout detection protocols only need to make conservative assumptions about the blackbox part, which means the protocol can use a shorter timeout interval to accelerate failure detection. Note that SafeTimer cannot make concrete suggestions about timeout interval: the user still has to estimate possible delays in the blackbox part. However, considering the various kinds of abnormal events in the whitebox part (Chapter 2.1), SafeTimer should be able to reduce timeout interval by at least tens of seconds.

Case studies. We present a few existing timeout detection protocols to show how SafeTimer models them and how they can benefit from SafeTimer.

Budhiraja et al. [9] discuss how to detect failures in primary-backup protocols, given different models. In the simplest model, which assumes clocks are sufficiently synchronized, links are reliable, and packet delay is bounded (δ), the sender can send heartbeats every τ seconds and the receiver reports a failure if it does not receive a heartbeat for $\delta + \tau$ seconds. SafeTimer can model this protocol in the following way: when the receiver receives a heartbeat at t, it creates a new interval from t to $t + \delta + \tau$ and checks whether it receives a heartbeat by the end of the new interval; the sender can define a successful heartbeat sending for interval i as sending a heartbeat at t_i and $t_i \leq t_{i-1} + \tau$. With the help of SafeTimer, this protocol may reduce δ because it does not need to include the delays of the whitebox part. This work also discusses more complicated models, which consider link failures and proposes a gossip protocol to route heartbeats through multiple links, which is adopted in Ceph. SafeTimer can model it accordingly. For example, to tolerate one link failure, the

sender can define a successful heartbeat sending as sending two heartbeats to two nodes by $t_{i-1} + \tau$. Similarly, SafeTimer may help to reduce δ.

In HDFS, a DataNode sends a heartbeat to the NameNode every three seconds, and the NameNode marks the DataNode as stale if it misses heartbeats for 30 seconds. In the common case, the NameNode will acknowledge a heartbeat to the DataNode; if the DataNode detects errors, it will send heartbeats more aggressively every second. SafeTimer can model it in the following way: when the receiver receives a heartbeat at t, it creates a new interval from t to $t + 30$ and checks whether it receives a heartbeat by the end of the new interval (note intervals can overlap in this case); the sender can define a successful heartbeat sending for interval i as 1) getting acknowledgement for one heartbeat or 2) sending heartbeats with an interval of less than one second. SafeTimer may help to reduce the 30-second interval because it does not need to consider delays in the whitebox part.

2.3 Design

SafeTimer enhances existing timeout detection protocols to tolerate long processing delays in the whitebox part. In this section, we first present SafeTimer's mechanisms and then prove its accuracy and completeness.

2.3.1 Accurate timeout at the receiver

As discussed in Section 2.2, SafeTimer assumes the application's heartbeat receiver defines multiple time intervals (interval i from $start_i$ to end_i), and reports a failure if no heartbeat is received during an interval.

SafeTimer guarantees that as long as the receiver's NIC has processed (either delivered or dropped) a heartbeat during interval i, SafeTimer's receiver module will not report a failure for interval i.

```
1   /* The application calls safetimer_check when
         missing heartbeats from start_i to end_i */
2   function safetimer_check(start_i)
3       send a barrier to itself
4       wait for barrier (with a timeout)
5       if barrier received and t_lastHeartbeat < start_i
6           read drop count in OS and NIC and reset to 0
7           if (drop count = 0 and t_drop < start_i)
8               return TRUE_FAILURE
9           else if (drop count != 0)
10              t_drop = current_time()
11          end
12      end
13      return FALSE_FAILURE

15  function safetimer_recv_thread()
16      when receiving heartbeat
17          t_lastHeartbeat = current_time()
18      when receiving barrier
19          notify safetimer_check
```

Figure 2.2: Pseudo code of SafeTimer's receiver module. For simplicity, it assumes there is only one sender, but it can easily be extended to support multiple senders. $t_{lastHeartbeat}$ records the timestamp of the last heartbeat. t_{drop} records the timestamp of the last drop event.

Its key idea is simple: if the receiver module does not receive any heartbeats by the end of an interval, it will check whether there are any pending or dropped heartbeats in its whitebox part, and if not, the receiver module can safely report a failure.

The key challenge, however, is how to implement this idea in modern OS. For fast packet processing, modern OS incorporates a highly concurrent design, which involves a pipeline with multiple threads in each stage. To identify whether some heartbeats are pending, a naive solution is to pause all threads and check all buffers, but this solution will have negative impact on performance and require intrusive modification to the kernel.

To solve this problem, SafeTimer incorporates a non-blocking design as shown in Figure 2.2: if the application does not receive any heartbeat by end_i, it will check whether any heartbeats are pending or dropped by calling *safetimer_check*, which sends a barrier

packet to itself (line 3). By crafting the barrier packet and configuring the system properly, SafeTimer ensures that a barrier will follow the same execution path of heartbeats. Therefore, if the receiver module receives the barrier, it can know that any heartbeats processed by the NIC before end_i must have been processed by the OS and SafeTimer as well, either delivered to the receiver module or dropped. We will present details about how to implement the barrier mechanism in Section 2.4. For now, the readers can simply assume SafeTimer somehow drives the heartbeats and the barriers into a FIFO channel.

If the receiver module receives the barrier, it will check again whether it has received a heartbeat ($t_{lastHeartbeat} < start_i$ in line 5). If not, the receiver module will read drop statistics from both the OS and the NIC: if $dropcount = 0$ and $t_{drop} < start_i$ (line 7), which means there are no drops in interval i, the receiver module can safely report a failure. If the barrier is dropped as well, the receiver module will not report a failure for interval i. In this case, the application will perform the same check in the following intervals and will eventually report a failure.

2.3.2 Stop sender when missing heartbeat

As discussed in Chapter 2.2, SafeTimer assumes that the application has rules to decide when to send heartbeats and whether they are sent successfully. In particular, without losing generality, SafeTimer assumes for each interval i, the application defines a deadline end'_i to send heartbeats, which should be earlier than end_i at the receiver side because of clock drift and network latency.

SafeTimer guarantees that if a sender cannot successfully send heartbeats by end'_i, the sender will not be able to send out any other packets after end'_i, because the receiver may report a failure at that time. This is necessary because the accuracy property requires that if

```
1   function safetimer_send_heartbeat(end'_i, end'_{i+1})
2       send heartbeats
3       if sending succeeded before end'_i
4           t_valid = end'_{i+1}
5       end

7   function safetimer_intercept_sending()
8       if (current_time() > t_valid)
9           drop the packet
10      else
11          perform the send
12      end
```

Figure 2.3: Pseudo code of SafeTimer sender module. The application defines end'_i as the deadline to send heartbeats for interval i; the application defines whether sending succeeds; SafeTimer maintains a timestamp t_{valid} to identify till when it is safe to send out packets.

the receiver reports a failure, the sender must have failed: violating this property can cause correctness issues. Taking the primary backup protocol as an example, a backup should only become active if the primary fails. If a backup receives a failure report and becomes active while the primary is still active, there will be two active nodes, creating a classic "split brain" problem [29].

Killing a sender when it is slow is not a new idea [12, 32], but how to implement it correctly despite arbitrary processing delays requires careful thought. Existing solutions ask a specific component (e.g., a watchdog [32]) to actively kill the sender. When considering arbitrary processing delays, however, such active solution is incomplete, because the delay of processing the "kill" command may allow the sender to be alive for an arbitrary amount of time, violating the accuracy property.

SafeTimer uses a passive solution by utilizing the idea of output commit [76]: a slow sender may continue processing, but as long as other nodes do not observe the effects of such processing, the slow sender is indistinguishable from a failed sender. As shown in Figure 2.3

(lines 3-12), SafeTimer's sender module maintains a timestamp t_{valid}, which indicates it is safe for the sender to send packets before t_{valid}. During startup, the sender sets t_{valid} to end'_0. If the sender successfully sends heartbeats for interval i, the sender extends t_{valid} to end'_{i+1} (line 4). Whenever the sender is about to send a packet, SafeTimer will compare the current time with t_{valid}: if current time is larger than t_{valid}, the sender will discard the packet (lines 7-12). Since heartbeat is blocked as well in this case, an invalid sender cannot extend t_{valid} and send packets in the future, unless with recovery operations.

Note that since the sending operation itself may take arbitrarily long, SafeTimer allows a packet generated before t_{valid} to be actually sent out after t_{valid}. This is fine because the packet is generated when the sender is still valid (i.e., when the receiver has not reported the failure).

2.3.3 Proof of accuracy and completeness

As discussed in Section 2.2, SafeTimer relies on the existing protocol to send and receive heartbeats in the blackbox part. When the existing protocol's assumptions about the blackbox part hold, we can prove that SafeTimer is accurate (i.e., never report failure for a correct node) despite arbitrary delays in the whitebox part and is complete (i.e., eventually report failure for a failed node) when the receiver does not experience slow processing or packet drops for sufficiently long. We provide the detailed proof in the appendix.

2.3.4 Benefit of SafeTimer

Because of the accuracy and completeness properties, the users of SafeTimer do not need to make conservative timing assumptions about the whitebox part. They do need to provide a reasonable estimation of such delay in the common case, because the sender needs some time to send out heartbeats. However, this requirement is only for performance: if the

actual delay is longer than estimation, which means the sender cannot send the heartbeat in time, SafeTimer will block the sender, which may cause unnecessary recovery and hurt performance, but this will not violate accuracy. Therefore, SafeTimer only requires the user to provide a reasonable estimation to make sure such events are *rare*. As a comparison, in existing protocols, if the actual delay is longer than estimation, system correctness can be violated, and that is why existing systems require conservative assumptions so that such events *never* happen. The gap between "rare" and "never" is where SafeTimer gains its benefit.

2.4 Implementation

This chapter presents the barrier mechanism at the receiver and the packet checking at the sender in detail.

2.4.1 Barrier mechanism at the receiver

The goal of the barrier mechanism is to ensure that if SafeTimer's receiver module sent a barrier to itself at t and received it later, then all heartbeats delivered by NIC before t must have been either delivered to the application or dropped. Achieving this property would be trivial if the OS processes all packets in FIFO order, but unfortunately, this is not true in modern OS. To illustrate the problem and motivate our design, we first present how Linux processes incoming packets.

Background. As shown in Figure 2.4, Linux incorporates a multi-stage pipeline to process incoming packets.

At the lowest level, an NIC buffers incoming packets in its RX queues and tries to transfer them to kernel's ring buffers: if the ring buffer has empty slots, the NIC will transfer

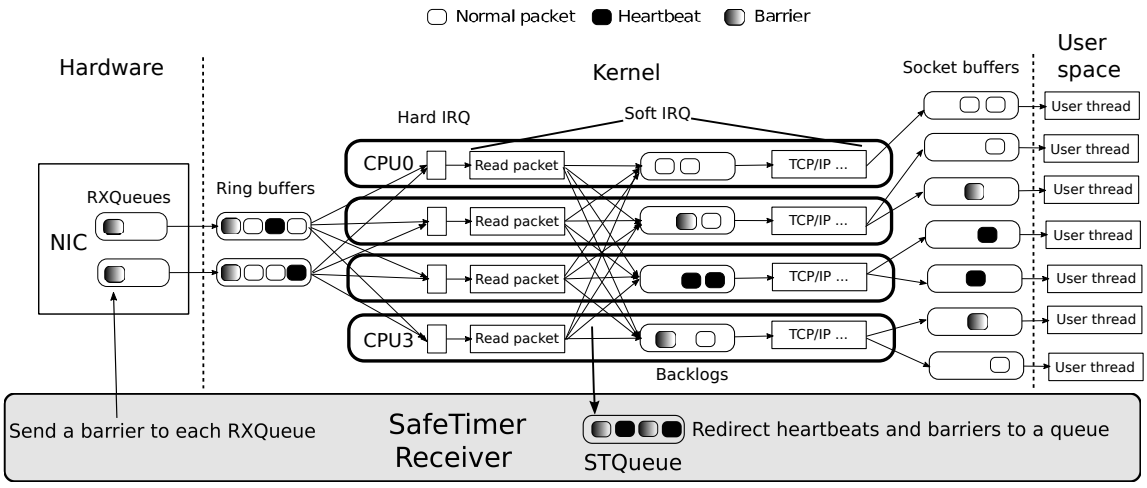

Figure 2.4: Barrier mechanism at the receiver. The algorithm in Figure 2.2 reads from STQueue.

the packet using DMA and fire an interrupt; if the buffer is full, the NIC will retry and may drop packets. For efficiency, modern NIC and Linux incorporate the Receive Side Scaling (RSS) technique [75] to allow parallel packet processing: the NIC creates multiple RX queues and the kernel creates an equal number of ring buffers so that each RX queue is mapped to a unique ring. Furthermore, Linux assigns a unique interrupt request (IRQ) number to each RX queue so that Linux can handle interrupts from different RX queues in parallel.

For efficiency, Linux separates interrupt handling into two parts—hard IRQ and soft IRQ—and invokes hard IRQ first. For an NIC interrupt, its hard IRQ simply sets some registers and triggers a soft IRQ. The soft IRQ reads packets from the ring buffer and executes the logic of the networking protocol, such as TCP/IP. The RSS technique allows Linux to handle IRQs in parallel.

By default, the soft IRQ reads from the ring buffer and executes the protocol logic within a single critical chapter protected by the lock of the ring. For more parallelism, Linux incorporates the Receive Packet Steering (RPS) technique [75]: when RPS is enabled, a soft IRQ reads a packet from the ring, puts it into a buffer called *backlog*, and then releases the lock of the ring. A separate thread, which may run on another CPU, will retrieve packets from the backlog and execute the protocol logic.

Finally the soft IRQ puts packets into socket buffers and the user-space threads may read from these buffers in parallel.

Such a multi-stage pipeline may re-order packets. Modern NIC and Linux preserve FIFO order for TCP packets with the same (sender IP, sender port, destination IP, destination port) and UDP packets with the same (sender IP, destination IP), by directing packets with same such information to the same RX queue, backlog and socket buffer. For SafeTimer, such guarantee is not enough since heartbeats and barriers are from different senders.

Overview of SafeTimer's solution. Our implementation is driven by three principles: 1) for portability, we hope to minimize modification to OS kernel code; 2) for performance, it should not incur significant overhead; 3) for portability, we hope to minimize dependence on specific NIC features or modification to NIC drivers.

As shown in Figure 2.4, SafeTimer re-directs heartbeats and barriers to a separate FIFO queue (called STQueue) early in the pipeline, so that they are not affected by re-ordering in later stages. However, since the earliest place we can perform such re-direction is after the soft IRQ reads the packets, RSS technique in the earlier stage may still re-order packets from different ring buffers. To solve this problem, SafeTimer sends a barrier packet to each RX queue/ring. If all of them later go through the STQueue, SafeTimer can know that all previous heartbeats are processed. The key to the correctness of this approach is that a soft

IRQ needs to grab the lock of the ring buffer when reading a packet from the ring, and thus packets from each ring are read in a FIFO manner. As long as SafeTimer re-directs a packet before the soft IRQ releases the lock, such per-ring FIFO order will be retained in the STQueue. Therefore, when SafeTimer retrieves a barrier from the STQueue, it knows all previous heartbeats from the same ring must have been processed.

Next we present each step in detail.

Forcing a barrier to go through NIC. SafeTimer requires a barrier packet to follow the same execution path of a heartbeat packet. Putting a barrier in the ring buffer does not work because the OS won't read from the buffer until an NIC interrupt is triggered. Therefore, SafeTimer receiver forces the barrier packet to go through its NIC. This task, however, is challenging for multiple reasons.

First, Linux has the loopback optimization to route a local packet by memory copy instead of sending it to the NIC. SafeTimer bypasses this optimization by sending the barrier directly to the device driver. This approach, however, creates a new problem: the NIC will actually send the packet to the router. To prevent loops, routing protocols usually have a constraint that a router should never forward a packet to the port where the packet is received. Therefore, the router will drop a barrier packet, whose destination and source are the same.

Our prototype uses an NIC with two ports and sends a barrier from one port to the other, which eliminates the above problem. This solution requires the receiver to have at least two links to the router, but considering the fact that redundant links are already widely used for fault tolerance, such requirement often does not incur additional cost. If redundant link is not available, another alternative is to use the virtual LAN (vLAN) technique to virtualize a physical port into two virtual ports [87].

Sending a barrier to a specific RX queue. A few NICs provide the "N-tuple filter" feature to direct packets to specified RX queues, which makes this problem trivial. However, we find this feature is not common so far [30]. Most NICs calculate a hash value based on the IPs and ports information in a packet and then direct the packet to an RX queue based on the hash value. Therefore, we propose a general solution based on the assumption that one cannot control which RX queue a packet is directed to, but packets with same IPs and ports will always be directed to the same RX queue.

SafeTimer uses a brute-force search approach: during initialization, its receiver module sends barriers with different sender ports to its NIC to see which RX queue they are directed to, until SafeTimer can find a port for each RX queue. Since usually there are not many RX queues, such procedure could finish quickly. The challenge, however, is how to know which RX queue (represented by its IRQ number) a packet is directed to. SafeTimer uses netfilter [74], which is a tool provided by Linux, to intercept soft IRQ functions to check whether a packet is a barrier, but soft IRQ functions do not carry the IRQ number of the RX queue. We can modify the driver to pass the IRQ number to the soft IRQ, but this violates our principle to minimize driver-specific modifications.

To solve this problem, we leverage the *irq-cpu affinity* configuration provided by Linux, which can configure the mapping between RX queues and CPUs during RSS. By default, it is configured to be an all-to-all mapping, which means any CPU can execute any IRQ to read from its corresponding RX queue/ring, but Linux also allows one-to-one mapping. We leverage this option to "test" whether a barrier is sent to a specific IRQ i: we map IRQ i to CPU 0 and the other IRQs to the remaining CPUs arbitrarily. When intercepting the soft IRQ function, SafeTimer reads the CPU ID: if the packet is a barrier and the IRQ function is

run on CPU 0, we can know the barrier must be sent to IRQ *i*; otherwise, SafeTimer tests a different *i* until it can find the right one.

Note that since the NIC always directs packets with same IPs and ports to the same RX queue, we only need to run the inferring procedure once for one machine. Afterwards we can use all-to-all mapping for efficiency.

Re-directing packets to STQueue. As shown in Figure 2.4, SafeTimer re-directs heartbeats and barriers to a FIFO STQueue after packets are read.

To implement this functionality, SafeTimer uses *netfilter* to hook the *ip_local_deliver* function, and configures iptable to re-direct heartbeats and barriers to a FIFO netfilter queue, which is called STQueue in SafeTimer. SafeTimer hooks *ip_local_deliver* because this is the earliest point packets can be re-directed in *netfilter*. SafeTimer sends heartbeats and barriers to specific ports so that they can be efficiently distinguished from normal packets.

This approach, however, is not fully correct when RPS is enabled: recall that when RPS is enabled, a soft IRQ will put a packet into the backlog and then releases the lock of the ring. In this case, *ip_local_deliver* is called after the lock is released and thus re-direction may not preserve the order of packets from the corresponding ring. To solve this problem, we use kretprobe [58] to intercept *get_rps_cpu* to return -1 for heartbeats and barriers: doing so essentially disables RPS for heartbeats and barriers. As a result, the re-direction will be executed under the protection of the lock of each ring and thus STQueue will preserve the order of packets from each ring. Normal packets, however, are not affected.

The timeout detection protocol (Figure 2.2) always reads heartbeats and barriers from the STQueue. However, SafeTimer does not remove heartbeats and barriers from later stages of the pipeline, because the OS needs to execute the logic of the network protocol, like congestion control or sending acknowledgements in TCP.

Reading drop count. SafeTimer's receiver module needs to read packet drop counts from both the OS and the NIC. Linux and most NICs have provided such statistics, but their implementation cannot achieve our goal.

In Linux, the NIC device driver periodically reads the drop count from the NIC, which can be fetched by reading /proc files system or using tools such as ethtool. Periodic reading means such statistics may be stale, which can cause SafeTimer's receiver module to miss recent drops and generate a false failure report. To make things worse, the NIC will reset drop count to 0 after it is read, so even if SafeTimer reads the drop count directly from the NIC, it may still get inaccurate results. To solve this problem, SafeTimer reads drop count from the NIC and then merges it with the number reported by the NIC driver. This is the only place SafeTimer requires modification to device drivers and OS kernel.

2.4.2 Blocking slow sender

As shown in Figure 2.3, SafeTimer's sender module blocks the sender if it cannot deliver heartbeats to the NIC in time. However, when sending a packet, Linux does not notify users whether or not the packet is delivered to the NIC successfully. Instead, it may write the packet to a buffer, return to the user, and send the packet to the NIC later, which may fail. To solve this problem, we use *kprobe* to intercept the function that the NIC driver invokes to reclaim resources after transmission is complete (e.g., *napi_consume_skb* or *__dev_kfree_skb_any*). As shown in Figure 2.3, SafeTimer applies the rules of the existing timeout detection protocol to check whether heartbeats are sent successfully. If so, SafeTimer's sender module will update t_{valid}. To block invalid packets, we use *netfilter* to intercept the *ip_output* function: if current time is larger than t_{valid}, the packet will be dropped.

Because of the processing delay, SafeTimer cannot get the exact time when a packet is sent. Instead, SafeTimer conservatively uses the timestamp after sending a packet, t_{after}: when checking whether a heartbeat is sent before end'_i (line 3 in Figure 2.3), SafeTimer compares t_{after} with end'_i. Such conservative approach ensures a sender failing to send heartbeats in time must be blocked, but it may also block a sender that has sent heartbeats in time, which is unnecessary but does not violate accuracy. Previous works have discussed how to minimize the impact of such unnecessary killing [64].

Since a slow sender process may communicate with other processes on the same machine, SafeTimer needs to block those processes as well, and thus it provides two blocking modes: the first blocks all processes on a machine; the second blocks only the sender process if the user is sure it does not communicate with other processes. Automatically tracking the information flow among different processes is out of the scope of this dissertation.

2.5 Evaluation

Our evaluation tries to answer three questions:

- What is the overhead of SafeTimer?
- Can SafeTimer achieve the expected accuracy property, despite long delays in the OS and the application?
- How much effort does it take to apply SafeTimer to existing systems?

To answer the first question, we have evaluated SafeTimer with a performance benchmark, which can send packets with different sizes, and compared its throughput and latency to those without SafeTimer. For the blackbox part, we use a simple protocol that sends heartbeats periodically with a configurable interval.

To answer the second question, we have injected long delays and packet drops at different layers at both the sender and the receiver to observe whether SafeTimer can prevent false failure report. Of course, this is by no means a complete test: we have proved the accuracy of SafeTimer in the appendix. This set of experiments serves as a sanity check about whether our implementation has actually achieved the expected properties.

To answer the third question, we have applied SafeTimer to HDFS and Ceph to enhance their timeout detection protocols and report our experience.

2.5.1 Overhead

SafeTimer incurs overhead for each packet at both the sender and the receiver: SafeTimer's sender module compares current time with t_{valid} before sending each packet; SafeTimer's receiver module re-directs heartbeats and barriers to the STQueue. To know whether a packet is a heartbeat or a barrier, the receiver module checks the destination port of each packet. When a sender fails, SafeTimer performs additional operations to block the sender, send barriers, and read drop counts, but since failure is rare, we focus on overhead in the failure-free case.

Since SafeTimer incurs overhead for each packet, such overhead should be relatively higher for workloads with smaller packets and thus we measure the overhead of SafeTimer with different packet sizes. However, TCP may merge small packets in the same connection and thus affect our experiment results. To prevent such effect, we use a ping-pong benchmark as suggested in a previous work [5]: we create multiple sender threads at the sender, each creating a connection to the receiver. The sender thread sends a packet to the receiver and waits for the receiver to forward the packet back. In this case, since each connection has

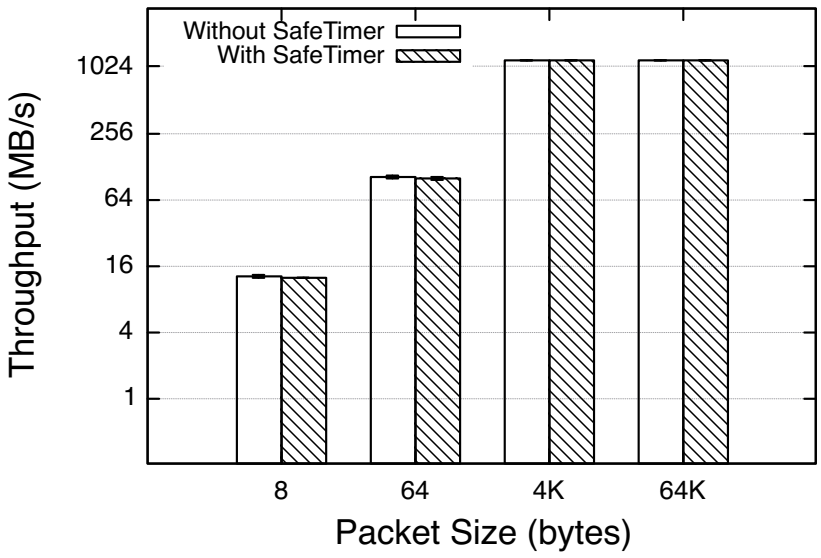

Figure 2.5: Throughput of the ping-pong benchmark with and without SafeTimer.

only one outstanding packet, TCP has no chance to merge packets. To increase load, we can increase the number of sender threads.

To measure the overhead of SafeTimer, we apply SafeTimer to the ping-pong benchmark and measure how it affects throughput and latency. To measure the maximal throughput, we increase the number of sender threads till we cannot gain higher throughput. To measure the latency, we run experiments under two loads: a light load of about 40% of the maximal throughput and a heavy load of about 90% of the maximal throughput. We do not measure the latency under the maximal throughput because in this case, the latency will be dominated by queuing delay. We run each setting 20 times to compute the average and standard deviation. We set the timeout interval of the blackbox part to be one second.

As shown in Figures 2.5 and 2.6, SafeTimer's overhead is small: for 4KB and 64KB packets, the overhead is less than 1%; for 8B and 64B packets, SafeTimer can increase p99 latency by 0.7% to 2.7% and decrease throughput by 1.6% to 2.4%. Such low overhead

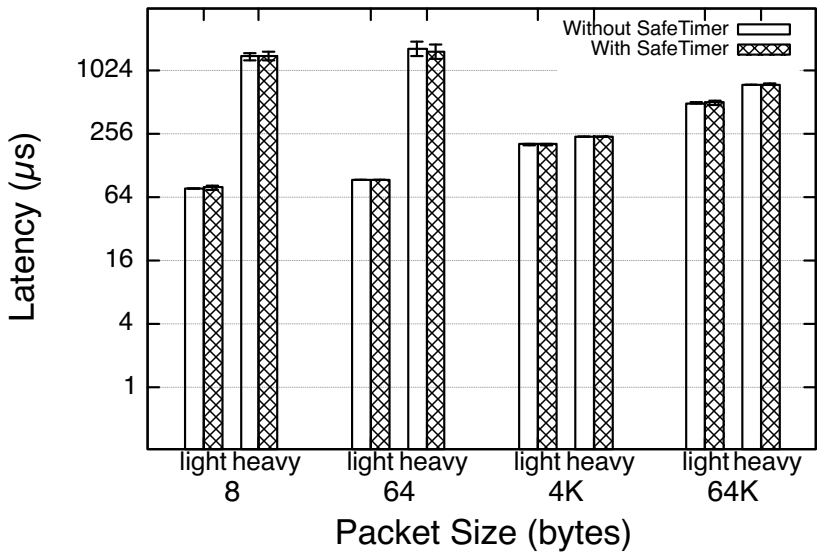

Figure 2.6: 99 percentile latency of the ping-pong benchmark with and without SafeTimer.

is reasonable because SafeTimer's additional work (i.e., comparing t_{valid} at the sender and reading destination port at the receiver) is small compared to other work the OS has to perform for each packet (e.g., interrupt handling, memory copy). To confirm the result, we run the same benchmark on another set of machines on CloudLab (m510 [25]) with different NICs (Mellanox ConnectX-3 10G) and we find the overhead of SafeTimer is similar.

2.5.2 Accuracy

Although we have proved the accuracy of SafeTimer, we hope to sanity check whether our implementation has achieved the expected property. For this purpose, we inject long delays and packet drops at different layers at the sender and the receiver. We compare SafeTimer to a vanilla timeout implementation, which has a user thread to periodically send heartbeats at the sender and a user thread to periodically check timeout at the receiver.

Node	Instrument Position	Injected Event	SafeTimer	Vanilla
Receiver	System call (recv)	Delay	No timeout	Timeout
Receiver	Socket (sock_queue_rcv_skb)	Delay/Drop	No timeout	Timeout
Receiver	NFQueue (nfqnl_enqueue_packet)	Delay/Drop	No timeout	N/A
Receiver	IP (ip_rcv)	Delay	No timeout	Timeout
Receiver	RPS (enqueue_to_backlog)	Delay/Drop	No timeout	Timeout
Receiver	Ethernet (napi_gro_receive)	Delay	No timeout	Timeout
Sender	System call (send)	Delay	Blocked	Alive
Sender	Socket (sock_sendmsg)	Delay	Blocked	Alive
Sender	IP (ip_output)	Delay/Drop	Blocked	Alive. Can observe drop.
Sender	Ethernet (dev_queue_xmit)	Delay	Blocked	Alive

Table 2.1: Verifying accuracy of SafeTimer by injecting long delay or packet drops. Gray cells indicate injection in kernel. N/A means this test case does not apply.

Table 2.1 summarizes the events we injected and how SafeTimer responds to these events. We inject long delays at all positions but only inject drops if the corresponding function can actually drop packets. In these experiments, we set timeout interval to be one second and inject a delay of two seconds. As shown in the table, SafeTimer correctly prevents false failure report at the receiver and blocks the sender in all cases. The vanilla implementation, however, violates accuracy in almost all cases except when a heartbeat is dropped in *ip_output*: in this case, the sender receives an error and can retry.

2.5.3 Case studies

To evaluate how much effort it takes to apply SafeTimer to real-world applications and its performance overhead, we have applied SafeTimer to HDFS [83] and Ceph [15].

APIs of SafeTimer. At the sender side, SafeTimer provides two APIs: *safetimer_send_HB* to send a heartbeat and check whether it is delivered to the NIC in time; *safetimer_extend* to extend the t_valid value. At the receiver side, SafeTimer provides one API: *safetimer_check* to check whether it is safe to report a failure.

HDFS. In HDFS, a DataNode needs to periodically send a heartbeat to the NameNode and if the NameNode misses a number of consecutive heartbeats, the NameNode will mark the DataNode as "stale".

We modified one line of code in NameNode's *isStale* function, which checks whether heartbeats are missing for a DataNode, to perform the additional *safetimer_check*. We modified six lines of code in DataNode to use SafeTimer's APIs to send heartbeats and check whether heartbeats are sent in time. To simplify modification, we do not remove HDFS' original heartbeat mechanism: this leads to duplicate heartbeats but during our experiments, the overhead is negligible.

We killed a DataNode and found the NameNode can correctly mark a failed DataNode as stale. We have measured the performance of an HDFS deployment with three DataNodes by using Hadoop's built-in benchmark tool DFSIO. We ran each experiment five times. Without SafeTimer, DFSIO can achieve a write throughput of 203 MB/s (stdev 12.6) and a read throughput of 627 MB/s (stdev 18.4); with SafeTimer, it can achieve a write throughput of 206 MB/s (stdev 5.5) and a read throughput of 632 MB/s (stdev 8.4). The difference is not statistically significant.

Ceph. In Ceph, an Object Storage Daemon (OSD) sends heartbeats to its two peers every 6 seconds and if they can't receive the heartbeat for 20 seconds, they will send a failure report to the Monitor, which will consider the OSD as failed if receiving two reports.

In this mechanism, an OSD is both the sender and receiver of heartbeats. We modified two lines of code in OSD's *heartbeat_check* function to perform the *safetimer_check* before sending the failure report; we modified five lines of code to use SafeTimer's APIs to send heartbeats and check whether heartbeats are sent in time.

We killed an OSD and found the Monitor can mark it as down. We have measured the performance of a Ceph deployment with three OSDs by using Ceph's inbuit benchmark tool RADOS. We ran each experiment five times. Without SafeTimer, RADOS can achieve a bandwidth of 43.3 MB/s (stdev 1.6); with SafeTimer, it can achieve a bandwidth of 42.2 MB/s (stdev 1.1). The difference is not statistically significant.

Chapter 3: Find Heterogeneous-Unsafe Configuration Parameters

While many distributed systems were initially designed under the assumption that all nodes share the same configuration, heterogeneous configuration has become increasingly popular for two reasons. First, heterogeneous hardware naturally calls for a heterogeneous configuration to achieve optimal performance [21, 44, 59, 66, 97]. Second, even for a homogeneous system, sometimes we need to change its configuration at run time to adapt to the workload, but rebooting the whole system with a new configuration may be too disruptive [7, 85]. To solve this problem, several approaches incrementally change the configuration of a subset of nodes, either by rebooting these nodes [23, 68, 73] or by utilizing application APIs [21, 40, 43, 65, 88], until all nodes have the new configuration. Both of these cases may cause different nodes to have different configurations, either in the long term or in the short term.

Heterogeneous configuration, however, may cause distributed configuration errors if not used properly. For example, if one node is configured to encrypt its communication channel while the other node does not decrypt the messages, then unsurprisingly the communication will fail. This type of errors is different from the configuration errors caused by invalid configuration values [3, 79, 92, 93, 96, 98]: in our case, both configuration values (i.e., using and not using encryption) are valid; the problem is caused by two nodes with different configurations communicating with each other.

The goal of this dissertation is to investigate, in real-world applications, which configuration parameters cannot be set in a heterogeneous manner. We call them *heterogeneous-unsafe configuration parameters* in this dissertation. To achieve our goal, we have developed an approach to identify such parameters.

At a high level, our approach is not much different from classic program testing: we test the target application with different heterogeneous configurations and different inputs to see whether the application will fail. However, this method also encounters the classic challenge of program testing: a particular configuration parameter may only take effect when a particular piece of code is executed; thus, to test whether the parameter is heterogeneous-unsafe, we need to drive the application to a potential corner case.

To address this challenge, we observe that mature applications usually have well-designed unit tests and integration tests, which have already considered this problem: to test the effects of a certain configuration parameter, some of these unit tests generate inputs so that the particular parameter will take effect and have rules to check whether the application is in a healthy state. Following this observation, we utilize existing tests including both unit and integration tests to find heterogeneous-unsafe parameters by assigning different configurations to different nodes in these tests.

We encounter two challenges when applying this idea. First, to run a unit test or an integration test with a heterogeneous configuration, we have to be capable of assigning different configuration values to different nodes. In the other words, we need to provide a unified configuration controller which is able to assign a specific parameter value to a specific node when running with either unit tests or integration tests. While this task is relatively trivial for integration tests where nodes can be identified with process ids, it is significantly more challenging in unit tests, which often create nodes as threads within a

process: on the one hand, a unit test may create a configuration object and share it with different nodes; on the other hand, a node may have subcomponents, which may create their own configuration objects. Both properties make it harder to map a configuration object to a particular node. To address this problem with minimal modification to the target application, we incorporate several heuristics to identify configuration sharing—in which case we clone the configuration object—and infer the mapping from configuration objects to nodes.

The second challenge is the large number of tests to run. To alleviate this problem, we incorporate several techniques: 1) we "pre-run" tests to identify which configuration parameters are used by each node type in each test, to avoid assigning a parameter to a node that will not use the parameter; 2) under the assumption that most parameters are safe, we introduce *pooled testing*, which tests several parameters together within one test, and separates them only if the pooled test fails.

Following these ideas, we have built *ZebraConf*, a framework to reuse existing unit tests and integration tests to find heterogeneous-unsafe configuration parameters. We have applied ZebraConf to Flink [1], HBase [2], HDFS [42], MapReduce [70], YARN [94] with their unit tests suites, and Cassandra [13] with its integration test suite. Our evaluation yields the following results:

- ZebraConf reports a total of 64 heterogeneous-unsafe configuration parameters in these applications. Our manual analysis shows 47 of them are truly unsafe parameters, and the remaining 17 are false positives. While many of these unsafe parameters are expected (e.g., parameters related to encryption, compression, and heartbeat), some of them are more subtle. For example, we find setting a heterogeneous bandwidth limitation on different DataNode instances in HDFS can cause one DataNode with a high limit to overload a DataNode with a low limit, so that the latter cannot send progress

reports in time, causing timeout. We further propose suggestions and workarounds to make a subset of these parameters heterogeneous safe.

- Regarding unit tests, with its heuristics, ZebraConf correctly maps configuration objects to different nodes in 89.3% to 98.4% of testing instances for each application . Achieving this level of correctness required adding or changing 21 to 38 lines of code to apply ZebraConf to each application.

- Pre-running tests and pooled testing reduce the total number of testing instances to run by two to four orders of magnitude for each application. As a result, all tests can finish within 6,012 machine hours. While this number is certainly not small, it is affordable since we can run these tests in parallel (we used up to 100 machines in our experiments) and they do not need to be run frequently.

3.1 The Goal and the Approach

3.1.1 Goal and Definitions

This work targets a distributed system, which is composed of multiple nodes (i.e. processes). We assume each node can be configured independently with its own configuration file.

To formally define heterogeneous-unsafe configuration parameters, we first introduce the following definitions:

- F denotes a configuration file and $F(p)$ denotes the value of parameter p in the configuration file.

- $HomoConf(F)$ denotes a homogeneous configuration, in which all nodes have the same configuration file F.

- $HeteroConf(F_1, ..., F_n)$ denotes a heterogeneous configuration, in which node i has configuration file F_i.

- I denotes a sequence of inputs to the target application. Inputs include explicit inputs through the application APIs, as well as all nondeterministic factors, such as timing and randomness, which are modeled as implicit inputs.

- A testing *oracle* can verify whether the application is in a correct state given I and either a $HomoConf(F)$ or a $HeteroConf(F_1, ..., F_n)$.

We assume that within any given node, all code always sees the same configuration. In other words, issues caused by incorrect implementation of online reconfiguration (e.g., missing updates to some variables depending on the parameter to be reconfigured) are outside the scope of our work.

Definition 3.1.1 (Invalid heterogeneous configuration)**.** *We say $HeteroConf(F_1, ..., F_n)$ is invalid if $\exists I$ such that*

$$\neg oracle(I, HeteroConf(F_1, ..., F_n)) \ \wedge$$
$$\forall_{i \in \{1,...,n\}} oracle(I, HomoConf(F_i))$$

Intuitively, this means that an invalid heterogeneous configuration is one that causes problems even if every configuration file is individually valid.

Definition 3.1.2 (Heterogeneous-unsafe configuration parameters)**.** *We say a set of parameters P is heterogeneous unsafe if*

- *there exists an invalid heterogeneous configuration $HeteroConf(F_1, ..., F_n)$ in which for every parameter in P, at least two configuration files have different values of the*

parameter, and all other parameters have the same value in all configuration files, i.e.,

$$\forall_p \left(p \in P \Longleftrightarrow \exists_{i,j} F_i(p) \neq F_j(p) \right)$$

and

- *P is minimal, i.e., there does* not *exist* $P' \subsetneq P$ *that satisfies the above condition.*

Intuitively, this defines the minimal set of parameters that may cause invalid heterogeneous configurations when given different values. If parameters do not depend on each other, this definition can further be simplified to be on individual parameters. The goal of this work is to identify heterogeneous-unsafe configuration parameters in real-world applications.

3.1.2 Our Approach

To understand whether certain configuration parameters are heterogeneous unsafe, we use the traditional software testing approach: we generate a number of heterogeneous configurations, each with different values of the target parameters; we then run the target application with these heterogeneous configurations and different inputs, and check whether the application encounters errors. However, the challenge of this approach is that a particular configuration parameter may only take effect when rarely executed code is executed, and thus when testing the parameter, we need to generate specific inputs to drive the application to the corner case.

To address this challenge, we utilize existing tests including both unit and integration tests built by the application developers: we run these tests with the corresponding heterogeneous and homogeneous configurations. Since the tests of a mature application should cover most of an application's code [86, 99], they naturally provide the ability to drive the application to corner cases and test whether the application is in a correct state. In other words, we assume

the tests can provide the input *I* and approximate the *oracle* in Definition 3.1.1. Following this idea, we have built ZebraConf, a framework to generate heterogeneous configurations, to run tests with these configurations, and to modify the target application to facilitate such testing.

Unit tests and integration tests. Traditionally, "unit tests" refer to tests that target individual functions or components of a system, and thus cannot be used for our purpose since they do not start multiple nodes. In contrast, "integration tests" refer to tests that target the whole system which will usually create a real cluster with mutiple processes for testing. However, to simplify testing, today's open-source software often implements its whole-system tests by running nodes as threads in one process and managing these tests as unit tests (e.g., MiniDFSCluster in HDFS, MiniCluster in Flink, etc). In this work, ZebraConf targets reusing both such whole-system unit tests and integration tests to find heterogeneous-unsafe configuration parameters. Regarding integration testing, for now, ZebraConf only supports the integration tests that are running on a single machines; however, extending ZebraConf to reuse integration tests running across multiple machines should not be a challenging task, though we have not done any experiments with those tests.

3.1.2.1 System Architecture

As illustrated in Figure 3.1, ZebraConf consists of three key components: TestGenerator, TestRunner, and ConfAgent.

At the top layer, TestGenerator determines which tests to run and what heterogeneous configurations to use for each test.

At the middle layer, given a test, either a unit test or integration test, and a heterogeneous configuration, TestRunner follows Definition 3.1.1 to test 1) whether the test reports an error

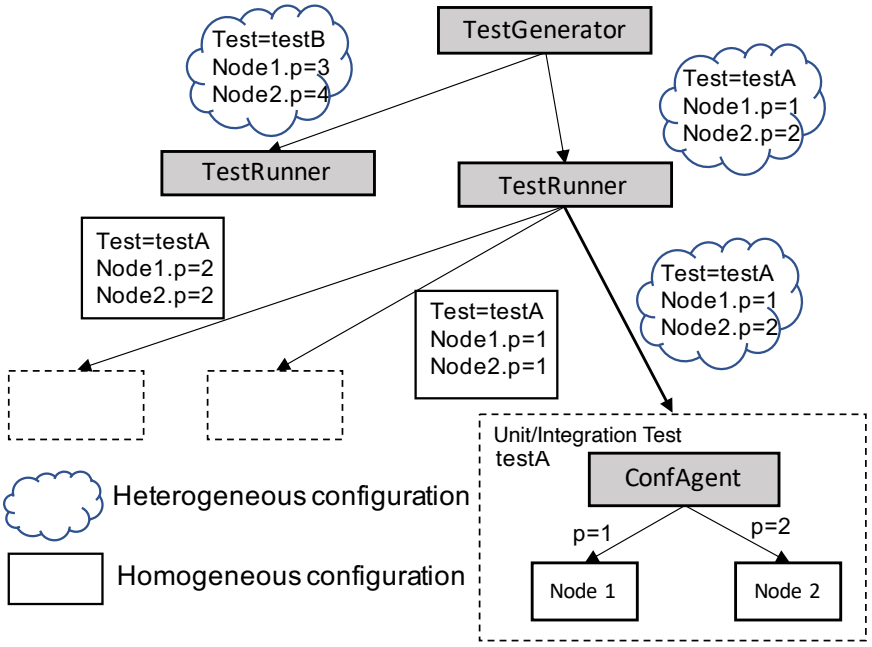

Figure 3.1: Overview of ZebraConf.

for the given heterogeneous configuration; and 2) whether the test reports an error for any corresponding homogeneous configuration.

At the bottom layer, ConfAgent is responsible for assigning specific parameter values to specific nodes during the test runtime based on the configuration assignment information specified by TestRunner. In ZebraConf, ConfAgent is designed and implemented as a unified parameter value controller that works with both unit tests and integration tests.

3.2 Design

3.2.1 TestGenerator

TestGenerator is responsible for generating all the test instances. In the general case, a test instance is represented by a tuple of a test and a heterogeneous configuration

HeteroConf$(F_1, F_2, ..., F_n)$. Table 3.1 shows the number of unit tests and parameters in different applications. As one can imagine, enumerating the combination of all parameter values and all tests will generate too many test instances. Exacerbating this problem, we observe that many whole-system unit tests and integration tests can take a long time (e.g., several minutes), because they need to wait for a cluster to be set up. To alleviate this problem, we introduce a number of strategies and techniques:

Test parameters independently. We assume that whether a configuration parameter is unsafe when set in a heterogeneous manner (i.e., different values on different nodes) does not depend on the values of other parameters. This assumption allows us to simplify Definition 3.1.2 to test each parameter individually rather than testing their combinations, which greatly reduces the number of test instances. With this strategy, TestGenerator converts the representation of a test instance into a tuple of 1) a test, 2) the name of the parameter to test, and 3) the parameter value at each node. All other parameters will use the original values in the particular test.

Of course this assumption does not always hold, and TestGenerator allows additional rules to specify that when testing parameter p_1 with value v_1, we should set p_2's value to v_2. Currently TestGenerator requires the developer's effort to generate these rules and in our experiments, we manually add rules for a few parameters which obviously depend on others. For example, in HDFS there is a parameter to configure whether to use the http or https protocol, and two parameters to set the http and https addresses. Following the HDFS documentation, we set the http address if using the http protocol and set the https address if using the https protocol. Future work could extract the relationship between different parameters automatically, by relying on parameter dependence analysis [20].

	Test Type	#Tests	# App-specific parameters
Flink	Unit Test	26,226	447 [34]
Hadoop Tools	Unit Test	1,518	N/A
HBase	Unit Test	4,985	206 [41]
HDFS	Unit Test	6,445	579 [46, 47]
MapReduce	Unit Test	1,423	210 [69]
YARN	Unit Test	4,806	465 [95]
Cassandra	Integration Test	740	178 [14]

Table 3.1: Statistics about the numbers of parameters and tests for each application we applied. Hadoop Tools provide a number of tools to support other applications, but do not have their own parameters. All other applications have their own parameters (see table's citations for details) and share the Hadoop Common library (see [37] for details), which has 336 parameters.

Select parameter values to test. For boolean parameters, selecting values is trivial since we only need to test true and false values. For other types of parameters, we manually select a few values that we believe are representative based on the documentation of the target application. For numerical values, apart from the default value, we select one that is much larger than the default value, one that is much smaller, and values that have specific meanings (e.g., 0 or -1 sometimes means this feature is disabled). For string values, we select the values listed in the documentation of the target application.

Select representative value assignment. If a test contains n nodes and we need to test a parameter with two different values v_1 and v_2, then there are 2^n ways to assign values to nodes. To reduce this number, we select a few representative assignment strategies, based on the observation that nodes of the same type are executing the same piece of code and thus are mostly symmetric. We first divide nodes into groups based on their types and then test each group G with the following strategies: 1) assign v_1 (v_2) to all nodes in G and assign v_2 (v_1) to all other nodes. This strategy tests heterogeneous configuration across different types

of nodes; 2) assigns values in a round robin order to nodes within G (i.e., assign v_1 (v_2) to the first node, assign v_2 (v_1) to the second node, assign v_1 (v_2) to the third node, and so on), and assign v_2 (v_1) to all other nodes. This strategy further tests heterogeneous configuration within nodes of the same type.

Pre-run profiling. The previous step generates a list of test instances, each specifying how to assign configuration values to different nodes in a test. However, not all these test instances are effective to test a heterogeneous configuration, and TestGenerator "pre-runs" all tests once to filter ineffective test instances.

First, although almost all integeration tests involve node creation, many unit tests do not create any nodes: as explained in Chapter 3.1.2, the term "unit test" initially referred to tests targeting individual functions and has changed recently to include whole-system tests. Unit tests that do not create any node are of course unable to test heterogeneous configurations. During the pre-run of a test, if the test does not start any node, then TestGenerator removes the test from its list.

Second, not all nodes in all tests use all parameters. This is also true for Integration tests. If we assign a parameter value to a node not using the parameter, then of course we are wasting time. This fact provides an opportunity for us to further trim the number of tests to run. To exploit this opportunity, during the pre-run TestGenerator records which node is using which parameter in each test. When generating test instances, TestGenerator applies the following rule: for a test with nodes of type A and a parameter p, TestGenerator will only generate test instances to test p on nodes of type A if these nodes actually use p in the pre-run. For example, in HDFS, TestGenerator will not test `dfs.datanode.balance.bandwidthPerSec` on NameNode because NameNode never uses this parameter.

A challenge of this technique is how to determine whether a node "uses" a parameter. In our current implementation, we define "use" as reading a parameter, which is easy to implement but is conservative since a node may read a parameter's value during initialization and never use the value later. Future work could explore using program analysis to improve the accuracy of identifying whether a parameter's value is used.

As shown in our evaluation (Table 3.5), pre-running profiling and filtering ineffective tuples allows us to reduce the number of testing instances to run by up to three orders of magnitude.

Pooled testing. To further reduce testing time, we observe that most configuration parameters are heterogeneous safe. This motivates us to use a divide-and-conquer approach: instead of testing only one parameter for heterogeneous safety when running a test, we test multiple parameters (called a "pool") together. If the test does not report any errors, then all of these parameters are assumed to be safe; otherwise, we divide these parameters into two groups and test each group recursively, until we can identify all unsafe parameters. In our evaluation, we set the maximal pool size to be equal to the number of parameters.

In order for this approach to be an effective optimization, we need most pools of parameters to be error free. However, we found that the efficiency of this approach is hampered by a small number of heterogeneous-unsafe parameters that fail almost every test. Examples include parameters related to encryption and compression, which are used by most tests. To solve this problem, if TestGenerator finds that a parameter has failed many tests, TestGenerator will mark the parameter as unsafe and avoid using it in future tests.

Test in parallel. For both unit and integration tests, they are independent from each other, which provides a natural opportunity to run tests in parallel. In our experiments, we run both unit and integration tests on a cluster of machines to reduce total wall-clock time.

3.2.2 TestRunner

TestRunner is responsible for running a test instance (a test and a heterogeneous configuration) generated by TestGenerator. Based on Definition 3.1.1, TestRunner will test both the heterogeneous configuration generated by TestGenerator and all corresponding homogeneous configurations: if the former one reports an error and the latter ones do not, then TestRunner will report a heterogeneous-unsafe parameter.

TestRunner's task is complicated by nondeterministic errors in tests which can happen in both unit and integration tests. For example, if a heterogeneous configuration has a probability to fail but does not fail in one test, then we may miss a heterogeneous-unsafe configuration parameter (i.e., false negative). If one of the homogeneous configurations has a probability to fail but does not fail in one test, then we may report a heterogeneous-safe parameter as unsafe (i.e., false positive). In our experiments, we find that false positives caused by nondeterministic errors are common.

To reduce false positives in the face of nondeterminism, we run multiple trials of a test instance (both its heterogeneous configuration and corresponding homogeneous configurations) until we can be sure that the parameter is heterogeneous unsafe with high probability, according to hypothesis testing using a significance level of 0.0001 (i.e., $1 - 99.99\%$).

To minimize run time, we run multiple trials of a test instance only if its heterogeneous configuration fails and none of its homogeneous configurations fail in the first trial. This

approach saves time but can result in false negatives due to nondeterminism. To reduce false negatives, a developer would need to run the test instances multiple times, which is not ideal but is the standard solution for most nondeterministic errors. On the other hand, although we run only one trial for most of the test instances, most parameters are tested by multiple test instances, reducing the chances of false negatives.

3.2.3 ConfAgent

In ZebraConf, ConfAgent is responsible for assigning specific parameter values to specific nodes during the test runtime based on the configuration assignment information specified by TestRunner. ConfAgent supports performing this task for both unit and integration tests. Since it is more challenging to support unit tests than integration tests, we will first describe the challenges and the solution for unit tests in Section 3.2.3.1 and 3.2.3.2, and then discuss how to support integration tests in Section 3.2.3.4. Finally, we will discuss the assumptions of applying ConfAgent to applications in Section 3.2.3.5.

Since the major challenge comes from heterogeneous configurations, our discussion focuses on this context, using an example shown in Figure 3.2.

3.2.3.1 Challenges

To run a unit test with a heterogeneous configuration, ConfAgent needs to be able to control the configuration values at each node. This would be trivial in a real distributed setting or in an integration test, in which each node would be running as a process: we could give each node a separate configuration file. However, the context of unit tests is significantly more challenging because unit tests often create nodes as threads within a single process, and all nodes inherently share the same configuration file, making it infeasible to assign different configuration files to different nodes.

```
/* Blank constructor */
public Configuration() {
  ConfAgent.newConf(this);
  ...
}

/* Clone constructor */
public Configuration(
    Configuration other) {
  ConfAgent.cloneConf(other, this);
  ...
}

/* Get the value of name
    parameter */
public String get(String name)
    {
  String value = properties.
      getProperty(name);
  return value;
  return
      ConfAgent.interceptGet(this,
      name);
}

/* Set the value of the name
    parameter */
public void set(String name,
    String value) {
  ConfAgent.interceptSet(this, name,
      value);
  ...
}
```

(a) Pseudocode of the Configuration class.

```
private Configuration conf;

/* Component init function */
public Component() {
  this.conf = new
      Configuration();
  ...
}
```

(c) Pseudocode of the Component class.

```
private Configuration conf;
private Component c;

public static void main() {
  ...
  /* create a blank config
      object */
  Configuration conf = new
      Configuration();
  Server server = new Server(
      conf);
  ...
}

/* Server init function */
public Server(Configuration
    conf) {
  ConfAgent.startInit(this,
      'Server');
  /* replace saving reference
      with refToClone */
  this.conf = conf;
  this.conf =
      ConfAgent.refToCloneConf(conf);
  /* initialize this Server */
  c = new Component();
  ...
  ConfAgent.stopInit();
}

protected void funA() { ...}
```

(b) Pseudocode of the Server class.

```
public void test () {
  Configuration conf = new
      Configuration();
  /* create servers */
  Server server1 = new Server(
      conf);
  Server server2 = new Server(
      conf);
  ...
  server1.funA();
  System.out.println(conf.get(
      XXX));
}
```

(d) Pseudocode of a unit test.

Figure 3.2: An example of how an application and its unit tests may use configuration objects, and how to modify the application to support ZebraConf with the ConfAgent APIs. The lines with suffix `ConfAgent` are added by the developer. If developers only want to reuse integration tests, then only two APIs highlighted with orange are needed.

To address this problem, we observe that well-designed applications usually keep track of configuration values in a dedicated configuration object (e.g., Figure 3.2a). The configuration object usually provides a *get* function to retrieve a certain configuration value and a *set* function to set the value. Therefore, if we can modify the configuration objects to return different values to different nodes, we will achieve our goal. This approach has the benefit that it only requires modifying a dedicated class. The challenge is how to determine which node is calling the *get* function of a particular configuration object. To illustrate the challenge, we describe a few approaches we tried that failed.

Determine caller based on configuration object. Initially, we thought that if each node uses one configuration object internally, then our task would be trivial: we could annotate the creation of each configuration object to connect it to a node. However, when investigating real applications and their unit tests, we found that this assumption was almost never true, manifesting in two different ways. First, a unit test often creates a configuration object by itself and then shares the object with different nodes. For example, as shown in Figure 3.2d, the unit test creates a `Configuration` object, and then uses the object to create two `Server` objects. In this case, these two `Server` objects and the unit test (the unit test itself is treated as a "client" node in ZebraConf) are sharing the `Configuration` object. In our experiments, we find configuration object sharing occurs in 99.9%, 99.8%, 96.5%, 100%, and 88.5% of the unit tests that involve configuration usage in Flink, HBase, HDFS, MapReduce, and YARN respectively. Second, sometimes a node creates multiple configuration objects, which violates our assumption as well. For example, at line 19 in Figure 3.2b, a `Server` object creates a `Component` object, which later creates its own `Configuration` object (line 5 in Figure 3.2c).

Determine caller based on object allocation chain. In our second attempt, we tried to tie each Java object to a node. Then if a Java object called `Configuration.get`, we could know which node was making the call. To implement this idea, we first annotated a few objects as roots, which are typically the main object of a node (e.g., DataNode and NameNode in HDFS). Then we applied the following rule: if object A's method creates object B, then A and B belong to the same node. While we found no correctness problems in this approach, it was too invasive: we needed to add a `node` field to each object; we needed to modify the constructor of each object to pass the `node` field from its creator; and we needed to do this not only for the target application, but also for any third-party libraries used by the application. As a result, this approach incurred too much overhead, in terms of both the effort to modify the application and the CPU and memory usage at run time.

Determine caller based on calling thread. In the third attempt, we implemented a simplified version of our second attempt by only keeping track of which node each *thread* belongs to. We annotated the main thread of a node as the root and applied the following rule: if thread A creates thread B, then A and B belong to the same node. Then whenever `Configuration.get` was called, we could retrieve the thread ID of the caller and infer which node was making the call. Compared to the second attempt, this approach was simpler since getting the thread ID whenever `Configuration.get` was called did not require tracking the object allocation chain from the thread to `Configuration.get`. However, this approach relied on the assumption that a node's code is only executed by its own threads, and once again, we found that the design of unit tests violates this assumption: for testing convenience, it is common for a unit test to directly call nodes' internal functions for various purposes, such as stopping a node, adding data, checking status, and injecting faults. As a result, a node's code may be called by the unit test thread (i.e., main thread), and thus we

cannot determine which node is calling `Configuration.get`. For example, in Figure 3.2d, line 7, the unit test calls an internal function `funA` of `server1`. In this case, `funA` should use the configuration object of `server1`, but determining the caller based on the calling thread would instead use the configuration object of the unit test.

3.2.3.2 ConfAgent's Solution

Application	Types of nodes
Flink	JobManager, TaskManager
HBase	HMaster, HRegionServer, ThriftServer, RESTServer
HDFS	NameNode, DataNode, SecondaryNameNode, JournalNode, Balancer, Mover
MapReduce	MapTask, ReduceTask, JobHistoryServer
Yarn	ResourceManager, NodeManager, ApplicationHistoryServer
Cassandra	CassandraDaemon, Client, Tool

Table 3.2: The types of nodes we investigated.

ConfAgent's solution is based on our first attempted solution (determining the caller based on the configuration object). To achieve high accuracy and to minimize the modification to the target application despite the two challenges (i.e., configuration object sharing across different nodes and multiple configuration objects within a node), ConfAgent incorporates several heuristics, based on our observation of how configuration objects are used in the unit tests and the applications.

Observation 1: the number of types of nodes is small. As discussed previously, for all methods, we need to define certain "root" classes or objects to separate different nodes at run time. Fortunately we find this is a simple task: all the applications we investigated have a well-defined node class for each type of node, e.g., NameNode and DataNode in HDFS.

The number of types of nodes is small, which means manual annotation is feasible. Table 3.2 records the node types we picked in our work.

Observation 2: flow of configuration objects. We observe that information about configuration objects can flow in three ways, providing hints about how to track them.

- *Observation 2.1: creating a new blank configuration object.* Both the unit test itself and a node may create new configuration objects (e.g., line 2 in Figure 3.2d and line 5 in Figure 3.2c). We observe that a node usually creates its configuration objects in an initialization function, typically the constructor function or another `init` function in the node class. This means we can annotate the initialization function to map a configuration object to its node. Note that sometimes the object is created by a function called by the initialization function, instead of by the initialization function itself (e.g., line 19 in Figure 3.2b calls the constructor of the `Component` class, which creates a new `Configuration` object). And sometimes a node may create multiple configuration objects, usually for its subcomponents. To capture such relationships, ConfAgent adds the following rule: if a configuration object is created at time t on thread A, and thread A executes the initialization function of a node between t_1 and t_2, and $t_1 < t < t_2$, then the configuration object belongs to the particular node.

- *Observation 2.2: cloning a configuration object.* Both the unit test and the application may clone a configuration object by creating an object with a constructor that copies values from an existing object. We observe that the original object and the cloned object usually belong to the same entity (i.e., a node or the unit test itself).

- *Observation 2.3: creating a new reference to a configuration object.* This will not create a new object, and thus will not affect the mapping from configuration objects to

nodes, except in the following case: we observe a node's initialization function often takes a configuration object as an argument and assigns it to an internal reference (line 16 in Figure 3.2b). In a real distributed setting, the main function of the node class will create the configuration object and use it to create the node class (lines 7–8 in Figure 3.2b), but in the unit test, the unit test itself replaces the main function and may share the configuration object with many nodes (lines 2–5 in Figure 3.2d). To solve this problem, we require the developer to replace such a configuration object reference in the initialization function with a clone (lines 16–17 in Figure 3.2b).

Observation 3: exceptions to the previous observations could lead to a high false positive rate. Exceptions to the previous observations may cause ConfAgent to fail to assign proper values to different configuration objects, leading to false positives. In particular, if ConfAgent assigns different values to configuration objects within the same node, it may cause errors that will not happen in a real distributed setting, since a node in a distributed setting will read the same value for the same parameter from a configuration file. Although the total number of exceptions is small compared to the total number of unit tests, the number of heterogeneous-unsafe configuration parameters is small as well, so these exceptions would lead to a high false positive rate if they were not filtered properly. ConfAgent tries to identify such cases during the pre-run: for each unit test, it tries to map each configuration object to either a node or the unit test itself; if it ultimately finds that a configuration object is mapped to no entity, then for this unit test, ZebraConf avoids testing any parameters used by the unidentifiable configuration object.

Based on such observations, ConfAgent works as follows: first, the developer needs to annotate the node initialization and configuration object creation with the API provided by ConfAgent (see Chapter 3.2.3.3). Then at run time, ConfAgent follows the following rules

(based on the observations above) to determine the mapping from configuration objects to nodes:

Rule 1.1: Configuration object creation (Observation 2.1). If a configuration object is created at time t on thread A, and thread A executes the initialization function of a node between t_1 and t_2, and $t_1 < t < t_2$, then the configuration object "belongs to" the particular node.

Rule 1.2: Configuration object creation. If a configuration object is created when no node has initialized, then we say that this object "belongs to the unit test."

Rule 2: Configuration object reference (Observation 2.3). If the developer replaces a configuration object reference with a clone during initialization, then the object to be cloned belongs to the unit test, and the cloned object belongs to the node that executes the initialization function.

Rule 3: Configuration object clone (Observation 2.2). If a configuration object is cloned from another configuration object but not by Rule 2, then these two objects belong to the same entity.

As mentioned previously, if ConfAgent fails to map a configuration object in a unit test with these rules during the pre-run, ConfAgent excludes the test instances that combine the unit test and the parameters used by the unidentifiable configuration object from further testing, since they may generate false positives. Our evaluation shows that for four out of the five target applications, less than 5% of the test instances are excluded; for the remaining one, about 10% of the instances are excluded. Such numbers indicate that these rules accurately cover a high percentage of unit tests.

3.2.3.3 ConfAgent API and Implementation

To implement the aforementioned rules, ConfAgent provides an API for the developer to annotate the source code of the target application:

- `startInit(node, nodeType)` and `stopInit()`. The developer should use these two methods to annotate the start and the end of the initialization function (e.g., lines 14 and 21 in Figure 3.2b). They serve to implement Rule 1.1.

- `newConf(conf)`, `cloneConf(origConf, newConf)`, and `refToCloneConf(origConf)`. These three methods are for tracking configuration objects, which are used to implement all of the rules mentioned above. The developer can annotate the constructor of the configuration class with `newConf` or `cloneConf` (e.g., lines 3 and 9 in Figure 3.2a). The developer should use `refToCloneConf` to replace a reference to a configuration object with a clone in the node initialization function (e.g., lines 16–17 in Figure 3.2b).

- `interceptGet(conf, paraName)` and `interceptSet(conf, paraName, paraValue)`. These two methods are for intercepting the get and set functions of configuration objects, in order to implement the heterogeneous configuration. The developer can place these two methods in the get and set functions of the configuration class (e.g., lines 17 and 22 in Figure 3.2a).

To implement these API methods, ConfAgent maintains the following data structures:

- A `nodeTable` object records the following for each node: `nodeID` is the hashCode of the corresponding node; `nodeType` is the type of the node (e.g., NameNode or DataNode); `nodeIndex` is *i* if the node is the *i*th node of nodeType, which is used by the TestGenerator to assign a configuration value to a particular node (note

that TestGenerator cannot use nodeID for this purpose since nodeID may not be consistent across multiple runs); `confIDs` is an array recording the hashCode of all configuration objects belonging to this node; `parentConfID` records the hashCode of the configuration object passed as the argument to the initialization function, if any.

- A `unitTestConfIDs` list records configuration objects belonging to the unit test. An `uncertainConfIDs` list records configuration objects that cannot be mapped to anywhere.

- The `parentTochild<childConfID, parentConfID>` map keeps track of all cloning relationships.

- The `threadContext<ThreadID, nodeID>` map keeps track of whether an initialization function of a particular node is executing on a thread.

When `startInit(node, nodeType)` is called, ConfAgent puts a new entry in `nodeTable` (confIDs is empty and parentConfID is null). It further puts the current thread ID and node ID in `threadContext`. When `stopInit` is called, ConfAgent removes the current thread ID from `threadContext`.

When `newConf`, `cloneConf`, or `refToCloneConf` is called, ConfAgent updates the information based on the rules mentioned previously:

- When `newConf(conf)` is called, if no node has initialized yet, ConfAgent puts `conf` in `unitTestConfIDs` (Rule 1.2). If `threadContext` has a pair of `<ThreadID, nodeID>` for the current thread, ConfAgent puts `conf` into `nodeTable` (Rule 1.1); otherwise, ConfAgent puts `conf` in `uncertainConfIDs`.

- When `cloneConf(origConf, newConf)` is called, ConfAgent searches if either `origConf` or `newConf` already belongs to a node (i.e., in `nodeTable`) or the unit test (i.e., in `unitTestConfIDs`): if so, ConfAgent puts the other one in the same group (Rule 3); otherwise, ConfAgent puts both configuration objects in `uncertainConfIDs`. In either case, ConfAgent puts the pair in the `parentToChild` map.

- When `refToCloneConf(origConf)` is called, ConfAgent firsts clones `origConf` into a new object `newConf`. ConfAgent then puts `newConf` in `nodeTable` with the `nodeID` retrieved from `threadContext`, and puts `origConf` in `unitTestConfIDs` (Rule 2). Furthermore, ConfAgent recursively searches `origConf`'s parent in the `parentToChild` map to move them from `uncertainConfIDs` to `unitTestConfIDs` (Rule 3). Finally, ConfAgent returns `newConf`.

As mentioned previously, if `uncertainConfIDs` is not empty at the end of a unit test during the pre-run, meaning that ConfAgent cannot properly map certain objects to a node, ConfAgent excludes the test instances that combine this unit test and any parameters used by the configuration objects in `uncertainConfIDs` from further testing.

When `interceptGet(conf, paraName)` is called, ConfAgent first searches whether `conf` is in `nodeTable`: if so, ConfAgent can retrieve its corresponding `nodeType` and `nodeIndex` and check whether TestGenerator has assigned a particular value to <`nodeType, nodeIndex, paraName`>, in which case `interceptGet` returns the assigned value. If `conf` is not in `nodeTable` or if TestGenerator has not assigned a particular value, ConfAgent returns the original value in `conf`.

ConfAgent utilizes `interceptSet` to solve the following problem: sometimes the unit test creates a configuration object with empty values, then creates a node with this configuration object, expecting the node to fill the empty values, and later retrieves these

values (e.g., line 8 in Figure 3.2d). Since we replace the reference with a clone, the unit test is unable to get the correct values after node initialization. To solve this problem, ConfAgent adds the following logic to `interceptSet(conf, paraName, paraValue)`: if `conf` belongs to a node in `nodeTable` and the `parentConfID` of the particular node is not empty, then ConfAgent updates the corresponding value of the parent `conf` as well.

To illustrate the whole workflow, we next present how ConfAgent works on the example shown in Figure 3.2.

Step 1: When the unit test starts, it creates a new blank configuration (line 2 in Figure 3.2d), which triggers ConfAgent's `newConf` function at line 3 of Figure 3.2a. At this moment, since no node has been initialized yet (i.e., `nodeTable` is empty), ConfAgent marks this configuration object as belonging to the unit test (Rule 1.2).

Step 2: The unit test creates `server1`: the constructor of the `Server` class triggers ConfAgent's `startInit` function (line 14 in Figure 3.2b). This function generates a `nodeID` for this `Server` object and then registers a new node with type "Server" and its `nodeID` in the `nodeTable`; it also registers `nodeID` (i.e., `server1`) in `threadContext`, indicating that the code of `server1` is running on the main thread.

Step 3: The `Server` constructor triggers the `refToClone` function of ConfAgent (line 17 in Figure 3.2b). This function first clones the object. Then it tries to assign the cloned configuration object to a node by searching the `threadContext` to find which node is running on the current thread. In our example, it finds `server1` is running, so it marks the cloned configuration object as belonging to `server1` (Rule 2). It also marks the configuration object to be cloned as belonging to the unit test, but since that object was already marked in Step 1, this step does not change anything.

Step 4: The `Server` constructor creates a component object, which creates its own blank configuration (line 5 in Figure 3.2c). It triggers ConfAgent's `newConf` function. In this case, since ConfAgent finds `server1` is running on the current thread from `threadContext`, it marks the new configuration object as belonging to `server1`.

Step 5: The `Server` constructor triggers ConfAgent's `stopInit` function (line 21 in Figure 3.2b), which unregisters `server1` from the `threadContext`.

Step 6: The unit test creates `server2` (line 5 in Figure 3.2d), which repeats Steps 2–5 and marks the corresponding configuration objects as belonging to `server2`.

Step 7: When the unit test calls `funA` (line 7 in Figure 3.2d), if this function calls `Configuration.get`, ConfAgent can intercept this function (line 17 in Figure 3.2a) and know that the configuration object belongs to `server1`. ConfAgent can manipulate the return value here to allow `server1` and other nodes to have different configuration values.

3.2.3.4 ConfAgent for Integration Tests

Compared to unit tests, supporting integration tests is less challenging. While ConfAgent has to track configuration object creation flow to help identify node entities, we can differentiate node entities directly process ids.

To enable ConfAgent to perform the task of assigning different parameters values for different nodes in integration tests, developers need to inject ConfAgent's `interceptGet` function at the `get` function of the configuration class (line 17 in Figure 3.2a), and the `startInit` function at the `initialization` function of a server class (line 14 in Figure 3.2c).

The `startInit` function will record the process id and the type of the node when it is being initialized. In addition, the `startInit` function will assign an incremental id for each

node based on its type. The current method to achieve this task is that, when a node is being initialized, ConfAgent will append a line of process id into a file and assign the number of lines as the id for the node. Currently, to prevent possible race condition, we inject a small period of sleep between each initialization of nodes and find race condition never occurs. A compelete solution requires using a lock to make the write and read operation synchronized between processes and we will implement it in the future.

3.2.3.5 Assumptions

Although we have tried to make our implementation generalize to different applications, it does rely on a few assumptions: 1) the application should have whole-system unit tests; 2) the application should have a well-defined configuration class, which contains all the configuration parameters used by the application; 3) for each type of node, a well-defined initialization function will initialize all its configuration objects, either by creating a new configuration object or by storing a reference to an argument of the function; 4) different nodes should not share any object that needs to read configuration values, since we cannot determine what value to give to a shared object; and 5) configuration objects are not stored as global variables, because that would prevent ConfAgent from classifying configuration objects.

Assumption 5 is only for unit tests, because configuration objects being stored as global variables will not prevent ConfAgent from reusing integration tests. A violation of assumption 2, 3, 4, or 5 does not completely prevent ConfAgent from working: we can always modify the source code to handle these violations, or skip certain parameters if they are shared. Of course the more violations the application has, the less effective ZebraConf becomes. In the evaluation chapter, we discuss our experience with violations of these assumptions in real-world applications.

3.3 Experimental Evaluation

We implemented ConfAgent with 689 lines of Java code, TestRunner with 636 lines of Java Code and 1,285 lines of shell script, and TestGenerator with 133 lines of Java code and 327 lines of shell script. We also have 492 lines of shell script to run tests with Docker containers [72].

Our evaluation tries to answer two questions:

- How many heterogeneous-unsafe configuration parameters can ZebraConf find in real-world applications? (§3.3.1)
- How can the individual techniques of ZebraConf help to improve its accuracy and reduce its running time? (§3.3.2)

To answer these questions, we have applied ZebraConf to Flink, HBase, HDFS, MapReduce, and YARN with their unit test suites, and Cassandra with its integration test suite. We manually analyzed all the reported problems to understand whether they are true problems or false positives. Our principles for separating true problems and false positives are as follows. 1) To check whether a failed test may happen in a real distributed setting to be a true problem, we check two properties: first, a client should not need to manipulate the private data of a server, which is only possible in a unit test, not in a real distributed setting; second, the error should not be caused by an inconsistent configuration within one node, which will not happen in a real distributed setting; 2) If the failed test causes an error in the application code, we classify it as a real problem. 3) If the failed test does not cause any errors in the application code, but violates some assertion in the test, we try to understand the assertions and make a best-effort determination whether the assertion would be meaningful

Parameter	Why parameter is heterogeneous unsafe
Flink	
akka.ssl.enabled	TaskManager fails to connect to ResourceManager.
taskmanager.data.ssl.enabled	TaskManager fails to decode peer message due to invalid SSL.
taskmanager.numberOfTaskSlots	JobManager fails to allocate slot from TaskManager.
Hadoop Common	
hadoop.rpc.protection	RPC client fails to connect to RPC servers.
ipc.client.rpc-timeout.ms	Socket connection timeouts.
HBase	
hbase.regionserver.thrift.compact	Thrift Admin fails to communicate with Thrift Server.
hbase.regionserver.thrift.framed	Thrift Admin fails to communicate with Thrift Server.
HDFS	
block.access.token.enable	DN fails to register block pools.
bytes-per-checksum	Checksum verification fails on DN.
blockreport.incremental.intervalMsec	End users may observe inconsistent number of blocks.
checksum.type	Checksum verification fails on DN.
replace-datanode-on-failure.enable	NN reports errors when Client tries to find other DNs.
client.socket-timeout	Socket connection timeouts.
datanode.balance.bandwidthPerSec	Balancer timeouts because DN fails to reply in time.
datanode.balance.max.concurrent.moves	Balancer becomes 10x slower due to DN congestion control.
datanode.du.reserved	End users may observe inconsistent size of reserved space.
data.transfer.protection	Sasl handshake fails between Client and DN.
encrypt.data.transfer	DN fails to re-compute encryption key as block key is missing.
ha.tail-edits.in-progress	JournalNode declines NN's request to fetch journaled edits.
heartbeat.interval	NN falsely identifies alive DN as crashed.
http.policy	Tool DFSck fails to connect to HTTP server.
namenode.fs-limits.max-component-length	Length of component name path exceeds maximum limit on NN.
namenode.fs-limits.max-directory-items	Directory item number exceeds maximum limit on NN.
namenode.heartbeat.recheck-interval	End users may observe inconsistent number of dead DNs.
namenode.max-corrupt-file-blocks-returned	End users may observe inconsistent number of corrupted blocks.
namenode.snapshotdiff.allow.snap	NN declines Client's request to do snapshot.
namenode.stale.datanode.interval	End users may observe inconsistent number of stale DNs.
namenode.upgrade.domain.factor	Balancer hangs due to block placement policy violation on NN.
MapReduce	
fileoutputcommitter.algorithm.version	Different output commit dirs cause Hadoop Archive error.
job.encrypted-intermediate-data	Reducer fails during shuffling due to checksum error.
job.maps	Reducer fails when copying Mapper output.
job.reduces	Reducer fails when copying Mapper output.
map.output.compress	Reducer fails during shuffling due to incorrect header.
map.output.compress.codec	Reducer fails during shuffling due to incorrect header.
output.fileoutputformat.compress	End users may observe inconsistent names of output files.
shuffle.ssl.enabled	NodeManager's Pluggable Shuffle fails to decode messages.

Table 3.3: The 46 true heterogeneous-unsafe configuration parameters found by ZebraConf. The prefixes of *dfs.* and *mapreduce.* for parameter in HDFS and MapReduce have been truncated in this table.

continued

Yarn	
http.policy	Client fails to connect with Timeline web services.
resourcemanager.delegation.token.renew	Users may observe newer tokens expire earlier than prior tokens.
scheduler.maximum-allocation-mb	ResourceManager disallows value decreasement.
scheduler.maximum-allocation-vcores	ResourceManager disallows value decreasement.
timeline-service.enabled	Client fails to connect to Timeline Server.
Cassandra	
cdc_enabled	Cassandra node fails to connect with peers schema disagreement.
receive_queue_reserve_endpoint_capacity	Cassandra node failes due to oversized inbound messages.
send_queue_reserve_endpoint_capacity	Cassandra node failes due to oversized inbound messages.
internode_socket_receive_buffer_size	Unbalanced data amounts on Cassandra nodes.
num_tokens	Cassandra node fails to restart due to mismatched token numbers.
internode_encryption	Cassandra node fails to handshake with peers.

in a realistic setting: if yes, we classify it as a real problem; otherwise, we classify it as a false positive.

Testbed. We run all experiments on CloudLab [24]. Each machine is equipped with two Intel Xeon 10-core CPUs, 192 GB DRAM, 480 GB SATA SSD (where we run experiments), and 1 TB SAS HD. We use up to 100 physical machines and allocate 20 and 5 Docker containers on each physical machine to run in parallel for unit tests and integration tests, respectively.

3.3.1 Heterogeneous-Unsafe Configuration Parameters in Real-World Applications

ZebraConf reports a total number of 64 heterogeneous-unsafe parameters in the fix target applications, and our manual analysis reveals 47 of them are true problems. We list all true problems in Table 3.3.

We categorize the true problems as follows, and we discuss them in the contexts of both long-term heterogeneous configuration and short-term heterogeneous configuration (i.e., partial reboot in a homogeneous system).

- Compression-, encryption-, authentication-, or transport-protocols-related parameters. These parameters affect the data format in a file or in a network communication, and thus if two nodes have different parameter values, one node will not be able to read data correctly. For a pair of nodes transferring data to each other, there is no reason to use heterogeneous values for these parameters in the long term. Reconfiguring these nodes may create a heterogeneous configuration in the short term, and a possible solution is to store the parameter value for each file or communication channel, so that a reconfiguration will not affect files or channels created before reconfiguration.

- Heartbeat-related parameters. If a heartbeat sender has a large interval value but the receiver has a small value, the sender may not send heartbeats in time, so that the receiver may decide the sender has died. While there is no good reason to use heterogeneous heartbeat intervals in the long term, it may happen in the short term due to the demand to reconfigure such values at run time. For example, since version 2.9.0, HDFS has supported reconfiguring `dfs.heartbeat.interval` at run time with its reconfiguration interface `hdfs dfsadmin -reconfig namenode` [43]. Such an online reconfiguration will create a short-term invalid heterogeneous configuration.

  We propose the following workaround: if the administrator needs to decrease the heartbeat interval, she should change the value at the heartbeat sender first, and then change the value at the receiver; if the administrator needs to increase the interval, she should change it at the receiver first and then at the sender. This strategy ensures that the sender interval is always less than or equal to the receiver interval, so that the receiver will not miss heartbeats. However, this workaround may not always work, since sometimes a node can act as both the sender and the receiver.

- Max-limit-related parameters. These parameters can encounter problems if we reconfigure a node's max limit to be smaller, while the state of the node already exceeds the smaller limit. The administrator should simply not try to reconfigure a node to decrease the max limit. In contrast, increasing the limit causes no problems in our experiments.

- Counts of tasks. Nodes with inconsistent values of these parameters will have problems retrieving data. These parameters should not be configured in a heterogeneous manner.

- Others. This group contains some interesting parameters that we did not expect to be unsafe. We provide details about some of them as follows.

dfs.datanode.balance.bandwidthPerSec. This parameter is used to specify the maximum amount of bandwidth that each HDFS DataNode can utilize for balancing. Starting in HDFS 0.20, developers made this parameter online reconfigurable with a new dfsadmin command, because administrators found "the optimal value of the bandwidthPerSec parameter is not always (almost never) known at the time of cluster startup" [44]. Setting `bandwidthPerSec` either too low or too high may bring the cluster into a "maintenance window," which is expensive for large clusters. Making this parameter reconfigurable at run time would help to avoid or alleviate this issue.

However, when setting this parameter in a heterogeneous manner, we observe the following problem: a DataNode with a high bandwidth limit may send many packets to a DataNode with a low limit so that the latter may run out of its quota. In this case, the latter will throttle its network traffic, which is expected. However, such throttling may prevent the DataNode from sending progress report to the Balancer, which is a tool to balance data in

the cluster. As a result, the Balancer times out eventually. To solve this problem, we propose that each node should reserve a small fraction of bandwidth for critical traffic like heartbeats or progress reports.

dfs.datanode.balance.max.concurrent.moves. A unit test reports timeout (100 s) when this parameter is configured to be 1 on DataNodes and 50 on the Balancer. The default value for this parameter is 50, which allows 50 threads on a DataNode to transfer blocks for balancing.

We checked the average balancing time for several configurations in the unit test: the time for (DataNode:50, Balancer:50) was 14 seconds and for (DataNode:1, Balancer:1) was 16.7 seconds—but the time for (DataNode:1, Balancer:50) was 154 seconds. While it made sense that (DataNode:50, Balancer:50) was faster than (DataNode:1, Balancer:1), it was initially unclear why (DataNode:1, Balancer:50) was significantly slower than (DataNode:1, Balancer:1).

Our further investigation showed that in (DataNode:1, Balancer:50), because Balancer is unaware of the 1-thread capacity on DataNodes, it still sends block transfer requests to DataNodes concurrently. However, a DataNode declines requests when its thread is already performing block transfer for balancing. When the request is declined, the corresponding Balancer dispatcher thread triggers a congestion control mechanism, which sleeps for 1100 ms before it retries. Since the DataNodes usually can finish a block transfer request within 1100 ms, such congestion control adds an extra delay to the whole procedure.

The reader may wonder why we want to set this parameter differently on Balancer and on DataNodes in the first place. Indeed, if all DataNodes have the same value, there is no good reason for the Balancer to use a different value. However, if different DataNodes have different values, then it is inevitable that the Balancer's configuration will be different

from some of the DataNodes. Because of the reported problem, it seems like a bad idea for the Balancer to use one value for this parameter. Instead, the Balancer should retrieve this value from different DataNodes, and accordingly send different numbers of tasks to different DataNodes. We observe that the community is already discussing this solution [45].

dfs.namenode.upgrade.domain.factor. This parameter is in effect when HDFS's block placement policy is set to `BlockPlacementPolicyWithUpgradeDomain`. *Upgrade domain* is a feature to support rolling upgrade, which upgrades a subset of DataNodes at a time. To minimize the chance of data unavailability, a rolling upgrade should affect at most one replica of a data block at a time. To satisfy this property, HDFS allows the administrator to divide DataNodes into groups, called upgrade domains, and HDFS ensures the replicas of a data block are placed into different upgrade domains.

A unit test that tests whether data rebalancing still honors a domain-aware block placement policy fails when Balancer and NameNode are configured with different numbers of upgrade domains. The rebalancing task never finishes because some block transfer requests are always declined by NameNode, which identifies the block transfer as an action that results in a violation of the placement policy being used.

Similar to the previous problem, if different NameNodes[1] have the same number of UpgradeDomains, there is no good reason for the Balancer to use a different value. If different NameNodes have different numbers of UpgradeDomains, however, it is inevitable for Balancer's configuration to be different from some HDFS NameNode's configuration. A possible solution for this issue is to let Balancer fetch the value of the domain factor from the corresponding NameNode, instead of reading from its local configuration file.

[1]Multiple NameNodes have been supported by HDFS since version 0.23.

dfs.blockreport.incremental.intervalMsec. This parameter determines whether, if a DataNode deletes a data block, the NameNode will receive the update immediately or whether the update can be delayed. If the DataNode is configured to use the delayed mode and the client's configuration file says a block deletion is reported immediately, a user may issue a delete command, expecting the block to disappear immediately, yet later find the block is still present (HDFS has one unit test simulating this case).

It is debatable whether this parameter presents a true problem. On the one hand, it does not cause any explicit errors in the application; on the other hand, it does expose an inconsistency to the user since the application's behavior does not match the configuration value. We find a total of 16 parameters having similar problems in our study. Our principle for separating true problems and false positives is that if the user can observe an inconsistency through the application's public APIs, then we mark the corresponding parameter as a true problem. If an inconsistency can only be observed through the application's private functions, we mark it as a false positive. For the 16 parameters that cause similar issues, this principle separates them into 7 true problems and 9 false positives.

Such problems show that it is often risky for an application user to make assumptions about the internal implementation of the application, and thus it may be better not to expose internals-related parameters to the application users.

Causes of false positives.

We summarize the top causes of false positives as follows:

- The setting does not happen in a real distributed system. Some tests check or manipulate the private data of a node, which cannot happen in a real system. For example, an HBase test directly opens a new region on HRegionServer by calling

`HRegionServer.openRegion`, with the client's configuration object. In a real distributed setting, an HBase client can only do so through an RPC, in which case the server will use its own configuration object.

- Violating assumptions. In the unit tests of Hadoop projects, different nodes share the InterProcess Communication (IPC) component, which has its own configuration object. However, the IPC component sometimes reads configuration values from external configuration objects as well. The combination of sharing the IPC component and the IPC component reading values from different places causes the IPC component to read different values in a heterogeneous test, which leads to false alarms for four IPC-related configuration parameters. After we modified one line of code in Hadoop to disable the sharing, the false alarms disappeared.

- Overly strict assertions. Many unit tests use assertions to check the state of the target application. While many of them are meaningful and reveal real problems, we find that a few are overly strict. For example, one test compares the image files of different NameNodes to ensure they are the same, which is meaningful, but it first unnecessarily compares the lengths of the two files. In a heterogeneous setting in which one NameNode compresses the image but the other NameNode does not, their image file lengths are different but their actual contents are still the same.

In our experiments, we find that false positives are usually not hard to identify, since unrealistic settings and strict assertions are usually explicit in the unit test code, which is usually short and easy to understand. The one exception is false alarms caused by IPC sharing, which took us one day to figure out. Understanding the reasons for true problems,

Application	Modified LOC
Flink	30 + 8
Hadoop Common	0 + 6
HBase	16 + 7
HDFS	24 + 6
MapReduce	12 + 6
Yarn	12 + 6
Cassandra	3 + 435

Table 3.4: Modified lines of code to apply ZebraConf to each application. The first number is lines related to modifying the node classes, and the second number is lines related to modifying the configuration class.

on the other hand, is a lot harder, since they often require a deep understanding of how the whole system works.

Regarding Cassandra where integration tests are being used, ZebraConf reports 6 true problems out of 7 reported parameters. `periodic_commitlog_sync_lag_block_in_ms` is reported as a heterogeneous-unsafe configuration parameter because system booting will timeout over a 120-seconds restrain set by the testing script when some nodes configured with 15 seconds (default value) while some nodes configured with 100 milliseconds. However, with a deep debugging of the homogeneous configuration of 100 milliseconds, we find the system booting time under this setting is also much slower than normal and will constantly cause timeout if we add one more node into the cluster. So, we regard this report as a false positive.

3.3.2 Effects of Individual Techniques

Effort to modify the applications. As discussed in Chapter 3.2.3.3 and Chapter 3.2.3.4, to use ZebraConf, the user needs to use ConfAgent's API to modify two types of class files: the node class and the configuration class. Table 3.4 shows the lines of code we needed to

Application	Original	W. prerun profiling	After removing uncertainty	W. pooled testing
Flink	7,193,881,080	2,019,422	1,972,278	259,573
Hadoop-Tools	373,850,400	356,016	346,588	89,744
HBase	557,761,680	6,145,374	6,033,174	1,438,929
HDFS	387,499,008	10,404,952	10,242,886	1,968,218
MapReduce	284,486,160	482,272	430,800	104,588
YARN	705,346,824	668,020	640,338	312,726
Cassandra	4,875,120	1,033,020	1,033,020	241,600

Table 3.5: The number of test instances generated after successively applied methods.

modify in each application. As the table shows, the modifications to support ZebraConf required low effort.

Among these applications, we find all applications have well-defined configuration class, except Cassandra. Unlike other applications where a configutation object is defined as a map where keys are parameters, in Cassandra, each parameter is defined as an individual public field. To modify Cassandra, we first add several generic `get` functions for each type of parameters, and modify the code such that every parameter read is replaced as a `get` invocation of the type of that parameter. This part of effort dominates the effort needed to hack the configuration class in Cassandra. We plan to use techniques such as JVMTI [55] that can help automate this part in the future.

HDFS, HBase, MapReduce, YARN, and Cassandra have well-defined initialization functions for each type of node. Flink is more complicated: its node class has initialization functions, which are used in a real distributed setting, but its unit tests do not invoke the initialization functions directly and instead copy the initialization code into the unit test code. It is unclear to us why Flink uses such a design; in any case, it required additional effort on our part to identify and annotate the copied initialization code.

Reduction of number of tests to run. Table 3.5 presents the effects of individual techniques incorporated by ZebraConf.

The first row is the number of test instances ZebraConf would run assuming the user has the same level of expertise as us but does not pre-run those unit tests as ZebraConf does. In particular, it assumes that the user tests each parameter independently, selects parameter values in the same way as us, and selects value assignment in the same way as us (Chapter 3.2.1). It also assumes the user knows which types of nodes the corresponding application includes, so she will not test nodes or parameters not included in the application. For example, for HDFS, she will not test RegionServer and related parameters, which belong to HBase; for HBase, however, she will test HDFS NameNode or DataNode and related parameters, because HBase depends on HDFS. As one can see in the table, even with these strategies, ZebraConf needs to run a large number of tests.

The second row reports the number of test instances after we pre-run the unit tests to filter ineffective ones. By looking at the test information, we observe that 1) many unit tests, which are designed to test individual data structures, do not even start any nodes, and thus are completely filtered by this step; 2) almost no unit tests use all parameters; 3) even for unit tests that use a certain parameter, in many cases the parameter is only used by a subset of nodes. Together these reasons allow us to reduce the number of test instances to run by up to three orders of magnitude.

The third row computes the number of test instances after we remove those with uncertain configuration objects (Chapter 3.2.3). As one can see, most of the test instances do not encounter uncertain configuration objects, confirming the accuracy of ZebraConf's approach to map configuration objects to nodes. Note however that although the percentage of tests

with uncertainties is small, if we did not remove them, they would yield a high false positive rate, because the percentage of unsafe parameters among all parameters is small as well.

The last row records the number of test instances ZebraConf actually runs after applying pooled testing, including both the executed pooled tests and individual tests (when a pooled test fails). As the table shows, pooled testing further reduces the number of tests ZebraConf needs to run.

These techniques reduce the number of tests to run by two to four orders of magnitude. With their help, ZebraConf is able to finish all tests within 6,012 machine hours. While this number is not small, it is affordable considering that an application does not need to be tested by ZebraConf frequently.

Effects of hypothesis testing. In our experiments, ZebraConf reported 2,248 test instances as failed in the first trial (i.e., the heterogeneous configuration test failed but all corresponding homogeneous tests succeeded), and hypothesis testing filtered 768 of the tests as false positives. These numbers have confirmed the necessity of hypothesis testing.

Chapter 4: Related Work

4.1 Distributed Systems and Timeout

Chandra et al. show that many classic problems in distributed system, such as consensus, can be solved with an accurate and complete failure detector [18]. In practice, timeout is widely used for failure detection, whose accuracy depends on their timing assumptions.

Synchronous systems. Under synchronous assumptions (i.e., delay of message transfer and clock deviation are bounded [18]), timeout can achieve both accuracy and completeness for failure detection. Many systems like primary-backup replication and HDFS [8, 9, 27, 35, 83] work under this assumption. To guarantee accuracy, these systems must make conservative assumptions about message delay and clock deviation. Previous works have tried to improve its accuracy by estimating the upper bound adaptively at runtime [6] and by killing a node if the failure detector reports the node has failed [12, 32]. ZebraConf can enhance synchronous systems to tolerate abnormal events in the OS and the application, without requiring any timing assumptions.

Asynchronous systems. Under asynchronous assumptions (i.e., delay of message transfer and clock deviation are unbounded), building a failure detector that is both accurate and complete is proved to be impossible [33]. Paxos [60,61,77] is a replication protocol designed for asynchronous environments: it is always correct (i.e., all correct replicas process the same sequence of requests) and is live (i.e., the system can make progress) when the

environment is synchronous for sufficiently long. Paxos is used as building blocks in larger systems like Spanner [26] and Microsoft Azure Storage [11]. Compared to synchronous replication systems, Paxos is more expensive in terms of number of replicas and messages. Asynchronous systems don't need accurate failure detection for correctness, but since there is a cost to recover a failure, ZebraConf may help to reduce such unnecessary recovery by reducing the number of false failure reports.

Lease systems. A number of systems [2, 19] install a replicated lease manager (e.g., Chubby [10] and ZooKeeper [51]): a server needs to acquire a lease from the lease manager before it can service clients; the server has to renew the lease before it expires, and if not successful, the server will stop servicing clients. For accuracy, this approach requires the clock speed of servers and the lease manager to be sufficiently close, but it does not require the delay of message transfer to be bounded. Lease systems strike a balance between cost and timing assumptions, but it has its own limitations: first, the centralized nature of the lease manager means if a long delay happens at the lease manager, all leases will expire and all servers will stop servicing, which does not violate the accuracy property, but is certainly undesirable. As a result, lease systems prefer coarse-grained leases [10], which hurts system availability as well, similar as using a long timeout. Second, the requirement of a replicated lease manager makes it less desirable in small-scale systems. Systems using leases can benefit from ZebraConf by installing its sender module to ensure a server will not continue servicing after its lease expires.

Failure detection without timeout. A few systems implement a failure detector without using timeout. For example, Falcon [64] and its following works [62, 63] install probes in routers to monitor servers and install probes at different layers in a server to monitor upper layers. This approach essentially converts the whole communication channel into a white

box. As a result, it requires intrusive modification to the routing layer, which makes its deployment challenging and sometimes impossible if the routers are out of the control of the user. To solve these problems, Falcon uses timeout as a backup.

Real-time OS.

Real-time Linux [78] and other real-time frameworks for Linux such as RTAI [82] and Xenomai [91] can guarantee important tasks or interrupts are scheduled before given deadlines. However, this is not sufficient to achieve our goal, because long delay is not only caused by untimely scheduling, but also caused by the fact that an important task is occasionally blocked by a heavy task (Section 2.1). Real-time scheduling can address the former problem, but not the latter one.

4.2 Heterogeneous Configurations in Distributed Systems

Heterogeneous configuration. While many distributed systems were initially designed under the assumption that all nodes have the same configuration (i.e., homogeneous configuration), heterogeneous configuration has become increasingly popular for several reasons.

First, heterogeneous hardware naturally calls for a heterogeneous configuration to achieve the best performance [21, 59, 66, 97]. For example, many systems allow the administrator to configure the number of threads, the size of memory, or the bandwidth limitation of each node, and such configurations naturally depend on the hardware setting of each node.

Second, even for a homogeneous system, it is often beneficial to reconfigure the system to adapt to the workload [21, 65, 68, 88]. While rebooting the whole system with a new configuration is always possible [4, 31, 49, 67, 100], it is often too disruptive especially for a large cluster. To solve this problem, recent works propose to incrementally reconfigure a

subset of nodes, either by rebooting these nodes (i.e., rolling restart) [7, 23, 68, 71, 73] or by utilizing application APIs [36, 40, 43, 57, 81], until all nodes are reconfigured.

For both cases, since it is often hard to determine the optimal configuration values when booting the system, a number of systems (e.g., Kafka [56], HBase [2], HDFS [42], and Redis [80]) provide APIs to allow the administrator to change certain configuration parameters of a node at run time. For example, HDFS parameter `dfs.datanode.balance.bandwidthPerSec` was made online reconfigurable starting from HDFS 0.20 [44]. The introduction of these APIs indicates a strong motivation to reconfigure nodes at run time, presumably leading to more heterogeneous configurations in the future.

While heterogeneous configuration is beneficial and arguably unavoidable, it may lead to correctness issues when nodes with different configuration values communicate. Some of these issues are obvious: if a node is configured to encrypt its data and another node is not aware that data is encrypted, they cannot communicate properly. Some of these issues are more subtle and perhaps unexpected as shown in our evaluation. A goal of this work is to find such heterogeneous-unsafe configuration parameters in real-world applications.

Finding configuration errors. A substantial amount of work targets identifying configuration errors caused by invalid configuration values (including but not limited to [3, 50, 79, 92, 93, 96, 98]). For example, ConfValley [50] defines a systematic way to validate configuration values. PCheck [93] extracts application code that checks the validity of configuration values and executes such code before deployment.

Our work is different from these prior works because a parameter can be heterogeneous unsafe even if every value at different nodes is valid (e.g., one node is configured to encrypt data and another node is configured not to encrypt data). Therefore, it is impossible to check such problems locally at one node.

Chapter 5: Conclusion

In this dissertation, I present two different approaches to mitiagte the problem of distributed configuration errors. To reduce the chance that distributed configuration errors can happen in heartbeat failure detection, I present a mechanism called *SafeTimer*, which enhances existing heartbeat failure detection protocols to tolerate long processing delays in the OS and the application layer. To find the heterogeneous configurations that can cause distributed configuration errors, I build a testing framework called *ZebraConf*, which uses the traditional software testing approach to test heterogeneous configurations. In the future, I plan to explore solutions to make using heterogeneous configurations safe in distributed systems.

Bibliography

[1] Apache Flink. `https://flink.apache.org`.

[2] Apache HBASE. `http://hbase.apache.org`.

[3] Mona Attariyan and Jason Flinn. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, Vancouver, BC, Canada, October 2010.

[4] Liang Bao, Xin Liu, Ziheng Xu, and Baoyin Fang. Autoconfig: Automatic configuration tuning for distributed message systems. In *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*, Montpellier, France, September 2018.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, CO, 2014. USENIX Association.

[6] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 354–363, Washington, DC, USA, 2002. IEEE Computer Society.

[7] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache Hadoop Goes Realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD'11)*, Athens, Greece, June 2011.

[8] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault Tolerance. *ACM Transanctions on Computer Systems*, 14(1):80–107, February 1996.

[9] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *CDCCA*, 1992.

[10] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[12] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[13] The Apache Cassandra Project. `http://cassandra.apache.org`.

[14] Cassandra Configuration Documentation. `https://cassandra.apache.org/doc/4.0/cassandra/configuration/cass_yaml_file.html`.

[15] Ceph. `https://ceph.com`.

[16] Ceph Default Heartbeat Configuration. `http://docs.ceph.com/docs/master/rados/configuration/mon-osd-interaction/`.

[17] Ceph MDS heartbeat timeout during rejoin. `http://tracker.ceph.com/issues/19118`.

[18] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[20] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*, Virtual Event, November 2020.

[21] Dazhao Cheng, Jia Rao, Yanfei Guo, Changjun Jiang, and Xiaobo Zhou. Improving Performance of Heterogeneous MapReduce Clusters with Adaptive Task Tuning. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):774–786, 2016.

[22] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[23] Cloudera Documentation - Rolling Restart. `https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/cm_mc_rolling_restart.html`.

[24] CloudLab. `https://www.cloudlab.us`.

[25] Hardware information in the CloudLab manual. `http://docs.cloudlab.us/hardware.html`.

[26] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. 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 *OSDI*, 2012.

[27] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, 2008.

[28] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. Understanding real-world timeout problems in cloud server systems. In *IC2E 18*.

[29] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: A survey. *ACM Comput. Surv.*, 17(3):341–370, September 1985.

[30] Overview of Networking Drivers. `http://dpdk.org/doc/guides/nics/overview.html`.

[31] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning Database Configuration Parameters with iTuned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[32] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, 52(2):99–112, February 2003.

[33] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[34] Apache Flink Documentation: Configuration. `https://ci.apache.org/projects/flink/flink-docs-stable/deployment/config.html`.

[35] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, 2003.

[36] Allow Configuration Changes without Restarting Configured Nodes. https://issues.apache.org/jira/browse/HADOOP-7001.

[37] Hadoop Hadoop Common Configuration File. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/core-default.xml.

[38] Hadoop Default Configuration. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/core-default.xml.

[39] Set the zk default timeout to 3 minutes. https://issues.apache.org/jira/browse/HBASE-3273.

[40] HBase-8544. Add a Utility to Reload Configurations in Region Server. https://issues.apache.org/jira/browse/HBASE-8544.

[41] HBase Configuration File. https://hbase.apache.org/book.html#hbase_default_configurations.

[42] Hadoop HDFS Project. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[43] HDFS-1477. Support Reconfiguring ‘dfs.heartbeat.interval‘ and dfs.namenode.heartbeat.recheck-interval without NN Restart. https://issues.apache.org/jira/browse/HDFS-1477.

[44] HDFS-2202. Changes to Balancer Bandwidth Should Not Require Datanode Restart. https://issues.apache.org/jira/browse/HDFS-2202.

[45] HDFS-7466. Allow Different Values for ‘dfs.datanode.balance.max.concurrent.moves‘ per Datanode. https://issues.apache.org/jira/browse/HDFS-7466.

[46] Hadoop HDFS Configuration File. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml.

[47] Hadoop HDFS RBF Configuration File. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs-rbf/hdfs-rbf-default.xml.

[48] HDFS Default Configuration. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml.

[49] Herodotos Herodotou and Shivnath Babu. Profiling, What-If Analysis, and Cost-based Optimization of MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[50] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. ConfValley: A Systematic Configuration Validation Framework for Cloud Services. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*, Bordeaux, France, April 2015.

[51] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, 2010.

[52] Internet control message protocol. `https://tools.ietf.org/html/rfc792`, 1981.

[53] InfiniBand Performance. `http://www.mellanox.com/page/performance_infiniband`.

[54] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. Hios: A host interface i/o scheduler for solid state disks. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 289–300, Piscataway, NJ, USA, 2014. IEEE Press.

[55] Java Virtual Machine Tool Interface (JVM TI). `https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/`.

[56] Apache Kafka. `https://kafka.apache.org`.

[57] Confluent Documentation: Dynamic Configurations. `https://docs.confluent.io/platform/current/kafka/dynamic-config.html`.

[58] Kernel Probes (Kprobes). `https://www.kernel.org/doc/Documentation/kprobes.txt`.

[59] Palden Lama and Xiaobo Zhou. AROMA: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC'12)*, San Jose, California, USA, September 2012.

[60] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[61] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.

[62] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 427–441, Lombard, IL, 2013. USENIX.

[63] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 9:1–9:16, New York, NY, USA, 2015. ACM.

[64] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *SOSP*, 2011.

[65] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R. Butt, and Nicholas Fuller. MRONLINE: MapReduce Online Performance Tuning. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*, Vancouver, BC, Canada, 2014.

[66] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous Configuration Optimization for Distributed Databases in the Cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*, Virtual Event, July 2020.

[67] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware'17)*, Las Vegas, Nevada, USA, December 2017.

[68] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. SOPHIA: Online Reconfiguration of Clustered NoSQL Databases for Time-Varying Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, Renton, WA, USA, July 2019.

[69] Hadoop MapReduce Configuration File. `https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml`.

[70] Hadoop MapReduce Project. `https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`.

[71] MAPREDUCE-442. Ability to Re-configure Hadoop Daemons Online. `https://issues.apache.org/jira/browse/MAPREDUCE-442`.

[72] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux journal*, 2014(239):2, 2014.

[73] MySQL 8.0 Reference Manual - Performing a Rolling Restart of an NDB Cluster. https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster-rolling-restart.html.

[74] Netfilter. http://www.netfilter.org/.

[75] Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[76] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proc. 7th OSDI*, November 2006.

[77] B. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly-available distributed systems. In *Proc. 7th PODC*, 1988.

[78] Linux Foundation Real-Time Linux Project. https://rt.wiki.kernel.org.

[79] Ariel Shemaiah Rabkin. *Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software*. PhD thesis, UC Berkeley, 2012.

[80] Redis. https://redis.io.

[81] Redis Commands: CONFIG SET Parameter Value. https://redis.io/commands/config-set.

[82] RTAI - the RealTime Application Interface for Linux. https://www.rtai.org.

[83] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[84] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Holzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in googleï£¡ï£¡s datacenter network. In *Sigcomm '15*, 2015.

[85] Daniel Sun, Alan Fekete, Vincent Gramoli, Guoqiang Li, Xiwei Xu, and Liming Zhu. R2C: Robust Rolling-Upgrade in Clouds. *IEEE Transactions on Dependable and Secure Computing*, 15(5):811–823, 2016.

[86] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Virtual Event, November 2020.

[87] Patricia Thaler, Norman Finn, Don Fedyk, Glenn Parsons, and Eric Gray. Media access control bridges and virtual bridged local area networks. Technical report, IETF, March 2013.

[88] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and Auto-Adjusting Performance-Sensitive Configurations. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, Williamsburg, VA, USA, March 2018.

[89] Yang Wang, Manos Kapritsos, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems. In *NSDI*, 2014.

[90] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *OSDI*, 2006.

[91] Xenomai - Real-time framework for Linux. `http://xenomai.org`.

[92] Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*, pages 265–280. USENIX Association, July 2020.

[93] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, USA, nov 2016.

[94] Hadoop YARN Project. `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[95] Hadoop YARN Configuration File. `https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-common/yarn-default.xml`.

[96] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. Context-based Online Configuration-Error Detection. In *2011 USENIX Annual Technical Conference (USENIX ATC'11)*, Portland, OR, USA, June 2011.

[97] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*, Amsterdam, Netherlands, June 2019.

[98] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOSâĂŹ14)*, Salt Lake City, Utah, USA, March 2014.

[99] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The Inflection Point Hypothesis: a Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, Huntsville, Ontario, Canada, October 2019.

[100] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*, Santa Clara, California, September 2017.