A FAIL-SLOW TOLERANT RAFT IMPLEMENTATION

BY

ANDREW YOO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Advisers:

Assistant Professor Tianyin Xu
Assistant Professor Shuai Mu, Stony Brook University

ABSTRACT

Fail-slow tolerance has been a long desired trait for computer systems. A fail-slow fault causes a hardware or software component to experience performance degradation without stopping or terminating.

We inject fail-slow faults into existing distributed database systems. We observe that they cannot tolerate fail-slow faults in even a minority of followers. To determine the root cause of this intolerance, we perform a comprehensive analysis on each database and categorize them into patterns. Every pattern is heavily connected to the implementation rather than the algorithm design.

We extend our own programming library, DepFast, that facilitates programmers to write fail-slow tolerant code, to account for these patterns. DepFast leverages coroutines and events to provide interfaces that minimizes slowness propagation. Using insights from our root cause analysis, DepFast also provides warnings to the user at runtime that inform the programmers of the patterns if they exist in the implementation. We build a fail-slow tolerant Raft implementation on top of DepFast and integrate it into a database (DepFastDB). DepFastDB can tolerate the same fail-slow faults injected into other databases. Furthermore, we inject the patterns into DepFastDB and show that our runtime analysis can detect these patterns with near-perfect accuracy in our trials.

"To my parents, for their love and support."

ACKNOWLEDGMENTS

Foremost, I want to thank Assistant Professor Tianyin Xu for being a "rock-star" advisor like his own advisor. When I initially started attending the University of Illinois at Urbana-Champaign, I always heard Tianyin discussing systems topics. Hearing his energy and excitement about the topic inspired me to pursue projects with the same commitment. One day, Tianyin requested that I work on a new project with him, which is the project that led to my thesis. I accepted immediately without question because I knew that I was going to work under an advisor that is dedicated to his work. Working with Tianyin has taught me lessons that I will treasure in my life. The most important lesson that Tianyin taught me was about aiming to become a winner and taking initiative. One of my biggest flaws going into UIUC was that I always seeked minimum effort to achieve a goal. Instead, Tianyin taught me that I should aim for my best effort and not settle for the bare minimum. One example that he provided was his disappointment with people who are boastful about submitting a paper that was rejected. While submission is a difficult task on its own, true winners will place the extra work to make sure the paper is accepted. I tried to incorporate this lesson into this thesis. The large amount of effort and sacrifices made to complete this thesis is thanks to Tianyin. Going forward after graduation, I hope to approach every project with the same mindset.

I want to thank my co-advisor, Assistant Professor Shuai Mu, for helping me work through the details of my thesis. First, Shuai taught me the value of setting ambitious goals. Without his vision for how distributed systems should be implemented, this exciting project would have never started. After all, becoming a winner at a small goal is not impressive. Second, Shuai is the most helpful advisor in terms of technical assistance that I have had the chance to work with. Shuai would sacrifice time out of his night to video chat with me to help me debug the code or discuss ideas for the project. His benevolence helped me greatly in this project. I will carry these traits into my own life.

I want to thank Professor Klara Nahrstedt for collaborating with me on a research project that is different from my thesis. Exploring ABR algorithms and multimedia systems helped me broaden my knowledge of systems and networking.

I want to thank the Siebel Scholars program for accepting me into their long history of accomplished and distinguished students. During the tough last year, Siebel Scholars allowed me to focus entirely on my work. I felt both humbled and motivated to be part of this program, and I aim to continue building on its legacy.

Finally, I want to thank students that have collaborated and discussed technical topics with me, including Eric Lee, Kuan-yen Chou, Jason Liu, Hung Tran, Ritesh Sinha, Yuanli Wang, and every student in Tianyin's lab. The experience has taught me that disagreements and conflicts are common in achieving goals and a profound understanding of topics, but dealing with them requires patience and maturity. I hope to share these experiences with them again in the future.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 MOTIVATION

Replicated state machine or RSM systems have become a critical part of modern infrastructures by laying a foundation for distributed databases [1, 2, 3, 4] and service management [5, 6, 7]. [1] An RSM system deploys distributed nodes that replicate data in a linearizable and fault-tolerant manner. Most of the fault-tolerance guarantees are provided by the design of underlying protocols [8, 9, 10].

Many RSM systems are based on Raft [9] that replicates a log or a series of commands on a majority of servers. Raft enforces a strong leader that sends log entries to other nodes or "followers". When a leader receives a client request, it will append an entry to its log and send an `AppendEntries` RPC to the followers in parallel. The leader will commit the log entry when it knows that a majority of the servers has replicated the entry. As a result, the protocol guarantees correctness as long as a majority of the servers are healthy.

These replicated state machine systems can effectively tolerate fail-stop faults in a minority of servers but fall victim to fail-slow faults. In a fail-slow fault, a software or hardware component experiences performance degradation without stopping or terminating [11, 12]. In RSM systems, a fail-slow fault in even a follower can propagate to the entire system.

Every major hardware component, such as memory, SSD, and NICs, is susceptible to such faults [12]. For instance, a firmware bug on the NIC can decrease the network performance to half [12]. While it is not a failure in the hardware itself, contention over hardware resources can also lead to a fail-slow fault. Contention with another memory-intensive process can cause a 40x degradation on a task [11]. Furthermore, software components can experience these faults through bugs and misconfigurations [13, 14, 15]. These faults such as contention over a shared resource such as a lock can cause delays in heartbeat messages and restart tasks on another node [15]. Misconfigurations in virtual-memory mapping can lead to about 50% performance degradation [11]. RSM systems implemented with Raft are not immune to such failures.

In our own experience, modern implementations contradict an important motivation be-

[1] This chapter reuses material from the following accepted paper: A. Yoo, Y. Wang, R. Sinha, S. Mu, and T. Xu, "Fail-slow fault tolerance needs programming support," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, Virtual Event, May 2021.

Authors are Andrew Yoo (author of this thesis), Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. According to the Authors' Rights page for the publisher (ACM), I as the author have rights to reuse the paper. The Authors' Rights can be found here: `https://authors.acm.org/author-resources/author-rights`. The DOI is yet to come as it was not published yet.

hind Raft [9] to create a consensus protocol that is easy to implement. On one hand, we do not expect Raft implementations to tolerate certain fail-slow faults by design such as a faulty leader. On the other hand, a minority of faulty nodes should not affect the system. In contrast, our measurement of real-world RSM implementations, which all use Raft or a similar protocol, exposes their inability to consistently tolerate fail-slow faults in even a single follower. In our experience, we quickly discovered that the underlying problems stem from the implementation instead of the design. To worsen the problem, these modern implementations are often ridden with spaghetti code and fragments scattered across different threads and callbacks despite the goals of Raft [9].

Our goal is to write a fail-slow tolerant Raft implementation by leveraging our form of programming support in the Dependably Fast Library (DepFast). DepFast is motivated by our observations of currently convoluted implementations of RSM systems. Therefore, DepFast facilitates developers to write distributed systems code that can tolerate fail-slow faults through expressive interfaces. DepFast focuses on faithfully tolerating fail-slow faults that the protocol should, such as a fail-slow follower. To achieve this, DepFast allows developers to control fail-slow points such that they can prevent slowness propagation. Built on this principle, DepFast provides an interface that does not wait on each event individually but on a group of events for a majority. However, this iteration of DepFast alone cannot demonstrate that it can account for the implementation problems found in the real-world RSM implementations.

To resolve these shortcomings, we must first understand the root causes in modern implementations that are leading to the lack of fail-slow tolerance. We perform a comprehensive analysis on three widely used databases with similar implementations that includes dissecting the symptoms and source code. We categorize each root cause into the following patterns. First, the leader will spawn repeated background tasks (network events that do not spawn from a client request) when there is a fail-slow follower. In TiDB, the leader will resend the same `AppendEntries` message to the slow follower more than twice upon receiving a response from the follower. Second, a fail-slow follower can lead to excessive backlogs on the leader. The leader in RethinkDB has an unbounded buffer that grows due to slow response from the follower. Third, transient performance problems become more apparent in the tail. In MongoDB, transient performance on the healthy follower will cause the system to experience a fail-slow fault on a majority of nodes for a brief period.

We extend DepFast by abstracting the patterns that we observed into our interfaces, analyzing the system at runtime, and sending warnings to the developers. First, we monitor the transitive dependency of each request to ensure that the number of background tasks does not excessively grow. Second, we incorporate the `Finalize` interface into our current library,

where the programmers must process dangling requests or free them to prevent backlog problems. Finally, we provide an expressive interface to track the extent to which the current system is susceptible to transient performance. These interfaces notify the programmers if they are falling into the same pitfalls as modern implementations.

We leverage the extended version of DepFast to implement DepFastRaft, a fail-slow tolerant Raft implementation, and integrate it into a distributed database, DepFastDB. We demonstrate that DepFastDB can effectively tolerate fail-slow faults that we found to degrade performance on other systems. DepFastDB only experiences performance degradation up to 5% range for more than 90% of measurements. Furthermore, we show that our runtime analysis can reliably detect the patterns in other databases by injecting them into DepFastRaft. Our runtime analysis has near-perfect accuracy across every pattern.

1.2 CONTRIBUTIONS

The contributions in the paper are as follows:

1. We inject fail-slow faults on three widely used RSM-based databases and analyze their performance. While the algorithm and design should tolerate fail-slow faults in a minority of nodes, implementation of the databases causes the databases to falter. We demonstrate that even a fail-slow follower can significantly affect performance.

2. We perform a comprehensive root-cause analysis on all three databases to discover what is causing the slowness to propagate. We characterize the root causes into patterns, provide lines of code, and analyze their impact on the systems. The three root causes in TiDB, RethinkDB, and MongoDB respectively are background tasks that are triggered repeatedly by message responses, backlog issues from sending requests to a fail-slow follower, and transient performance.

3. We briefly detail the main components of the Dependably Fast Library (DepFast) that facilitates implementation of fail-slow tolerant systems. We then extend DepFast to abstract the patterns into our interfaces, analyze the system at runtime, and send warnings to the programmers. Using DepFast, we build our fail-slow tolerant Raft implementation, DepFastRaft.

4. We integrate DepFastRaft into a replicated key-value database, DepFastDB. Our evaluation is two-fold. First, we demonstrate that DepFastDB can tolerate fail-slow faults on a minority of followers. Second, we simulate the root-cause patterns by creating

flawed variants of DepFastRaft and show that DepFast can reliably detect the implementation flaws.

CHAPTER 2: MEASUREMENT

2.1 TARGET SYSTEMS AND CONFIGURATIONS

Systems As motivation for our work, we evaluate the fail-slow fault tolerance of three mature RSM-based distributed systems (TiDB [16], RethinkDB [17], and MongoDB[18]). [1] RethinkDB and MongoDB both utilize a single Raft consensus group, which has a leader and multiple followers. Meanwhile, TiDB leverages the MultiRaft architecture, which separates data into ranges that have their own consensus group. As a result, each node could be a leader and a follower of different consensus groups. However, these ranges still maintain the principles of Raft consensus, such as availability and prevention of data loss with a minority of failures. Therefore, the systems should handle up to to a certain number of fail-slow faults seamlessly in principle. Specifically, a single fail-slow follower should not hinder performance.

Configuration We configure the databases to enforce strong consistency if it is not the default configuration. We also disabled chained replication [19] because it causes slowness propagation from a fail-slow fault in a follower by design [20]. In our preliminary testing, we found that chained replication worsens the performance degradation from a fail-slow follower. Our focus is on the implementation level instead of the design level.

Deployment We deploy the databases on the Azure cloud. Each `Standard_D4s_v3` virtual machine instance has 4 CPUs, 16GB RAM, and a 64GB SSD for data.

2.2 FAIL-SLOW FAULTS

Fail-Slow Faults We build a framework to inject fail-slow faults on different major system components into the widely used databases. We selected these fail-slow faults based on prior work in the area [12, 14]. Table 2.1 displays the faults that we have injected.

[1] This chapter reuses material from the following accepted paper: A. Yoo, Y. Wang, R. Sinha, S. Mu, and T. Xu, "Fail-slow fault tolerance needs programming support," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, Virtual Event, May 2021.

Authors are Andrew Yoo (author of this thesis), Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. According to the Authors' Rights page for the publisher (ACM), I as the author have rights to reuse the paper. The Authors' Rights can be found here: `https://authors.acm.org/author-resources/author-rights`. The DOI is yet to come as it was not published yet.

Fail-slow type	Fault injection
CPU (slow)	Use `cgroup` to limit each RSM process to utilize only 5% CPU
CPU (contention)	Run a contending program (assigned with 16× higher CPU share than the process)
Disk (slow)	Use `cgroup` to limit disk I/O bandwidth available for the RSM process
Disk (contention)	Run a contending program that writes heavily on the shared disk
Memory (contention)	Use `cgroup` to set the maximum amount of user memory for the RSM process.
Network (slow)	Add a delay of 400 milliseconds to the network interface using `tc`

Table 2.1: Fail-slow faults used for measuring fail-slow tolerance of existing RSM databases (MongoDB, TiDB, and RethinkDB) and DepFastDB.

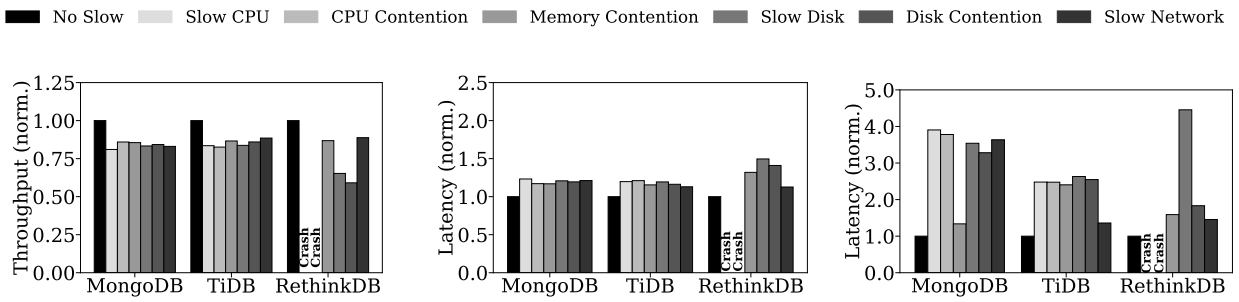
Fault Injection We set up a three-node deployment in which every node is a VM instance. Then, we inject a fail-slow fault on a single follower. Since MongoDB and RethinkDB have dedicated leader and follower nodes, we select one of the two follower nodes to throttle. However, TiDB leverages a Multi-Raft architecture that allows each node to host a leader and follower (§2.1). We modify the configuration of TiDB such that every node hosts only leaders or only followers.

2.3 WORKLOADS

For our workload, we leverage Yahoo! Cloud Serving Benchmark (YCSB) [21]. We initially start with two healthy followers on three-node deployments on the three widely-used databases. Afterwards, we inject fail-slow faults from Table 2.1 into a follower. The workload is comprised of 100% writes as a write will need to be replicated on a majority of nodes. Depending on the database, we run 256-1200 concurrent clients, which causes the leader to experience up to 75% utilization. We measure the performance on throughput, average latency, and P99 tail latency.

2.4 RESULTS AND FINDINGS

Figure 2.1 presents the performance of the the three databases after injecting a fail-slow fault on a follower node. Each database have different absolute performance. Therefore, we



(a) Throughput (b) Average Latency (c) P99 Latency

Figure 2.1: Performance of three RSM databases with a fail-slow follower (different type) in three-node setups.

normalized the performance numbers based on the performance of the experiment with no fail-slow followers.

In every type of injected fail-slow fault, there is nontrivial performance degradation in all metrics. The results demonstrate that a single fail-slow follower can decrease throughput by 17-41%, average latency by 21-50%, and P99 tail latency by 1.6-3.46x. Consequently, none of the databases are fail-slow tolerant even when we throttle a single follower.

2.5 DISCUSSION

In our process of analyzing slowness propagation, we discovered the challenges of debugging the root causes in the databases. Finding the root causes in the implementations pertaining to their performance in §2.4 consumed a total of two person-years. The method involved a binary search by minimizing smaller and smaller fragments of code using timestamping. In a vacuum, a binary search should be a simple and straightforward process. In reality, the process is complicated by spaghetti code where fragments are scattered across the implementation. Pinpointing the critical point of the code in the implementation and understanding how components interact is complex.

In our collaboration with developers of two of the databases, even they have admitted to lacking knowldedge of how the slowness propagates. Even with the same method of timestamping, they can rarely find the root cause. In fact, the developers do not fully understand every part of the system and cannot describe the whole system end to end.

Problem with Programmers We can assume several reasons to explain the difficulty of debugging the root causes of the slowness propagation. First, asynchronous programming with callbacks has been lauded as the primary way of developing distributed systems over

the synchronous alternative [22]. In popular asynchronous event-driven libraries such as `libev` and `libuv`, programmers utilize a message loop that processes messages and executes callbacks. Second, it is intuitive to program many distributed algorithms such that the system takes an action upon receiving a message as many such papers describe them in that fashion.

Combining this pattern with asynchronous programming can result in spaghetti code. For instance, a Paxos system progresses through 3 phases that have a minimum of 3 callbacks. If they function in a 5-replica system, the total number of callbacks extends to 15. Disk logging can double the callbacks. Monitoring these callbacks becomes even more cumbersome to debug as demonstrated in our own experience.

Logic vs Utility Our observation implies that it is important to differentiate logic, such as the Raft logic, from the utility, such as disk operations, in a clear abstraction. First, recognizing the origin of the problem between logic and utility can save valuable time in debugging the propagation. Since a root cause from utility is easier to fix, knowing that the root cause is not related to logic can be helpful. Second, this abstraction can increase transparency between the two parts. Without knowledge from the logic end, the utility must execute tasks without specific optimizations for fail-slow faults. When the Raft logic broadcasts `AppendEntries` messages to all replicas, the utility part will send the message to every replica without any questions or conditions. As we have discovered, this pattern has caused backlog problems where the buffer in RethinkDB continued to increase. If the logic can communicate with the utility, the utility aspect could adjust by dropping messages for a slower connection.

Our Solution The experience motivates us to explore a solution that places fail-slow faults as the highest priority in the fault-tolerance implementation on distributed systems. We aim to separate the fail-slow components from other parts of the code to minimize slowness propagation stemming from a fail-slow fault. We achieve this goal by designing a programming framework that monitors and handles fail-slow faults.

CHAPTER 3: ROOT CAUSES

We observed the following root cause patterns. [1] First, fail-slow faults can propagate through repeated background tasks (events that are not generated directly by a client request). For instance, the leader in TiDB sends the same `AppendEntries` message more than twice to a fail-slow follower that is triggered by a response to an earlier message. Second, a fail-slow follower can result in excessive backlogs inside the leader, which can increase processing overhead and consume resources. In RethinkDB, an unbounded buffer on the leader can grow as a result of a fail-slow follower. Third, fail-slow faults can enlarge the effects of transient performance issues. In a three-node deployment, a fail-slow follower will cause transient performance of the other follower to propagate. We confirmed the aforementioned issues with the developers.

3.1 REPEATED BACKGROUND TASKS PROBLEM

3.1.1 Overall Description

Some systems spawn an arbitrary number of background tasks resulting in increased network and disk activity when there is a fail-slow follower. We define background tasks to be any messages that do not originate from a client request. TiDB has an example of this behavior as `MsgAppend`, which we will call `AppendEntries` messages to adhere to Raft terminology, may be transmitted due to responses from heartbeat or previous `AppendEntries` messages to the follower. When the leader in TiDB receives such responses, it will check certain conditions to determine whether to retransmit a previous `AppendEntries` message. These conditions are more likely to be satisfied when we add slowness to a follower. Under a fail-slow fault, the leader will not only send the same `AppendEntries` message twice but also retransmit up to 30 times.

We describe three key fields of progress objects maintained in the leader for each follower that become problematic with a slow follower. First, the progress object has sliding windows called `Inflights` for each follower to track the number of currently outbound messages.

[1] This chapter reuses material from the following accepted paper: A. Yoo, Y. Wang, R. Sinha, S. Mu, and T. Xu, "Fail-slow fault tolerance needs programming support," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, Virtual Event, May 2021.

Authors are Andrew Yoo (author of this thesis), Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. According to the Authors' Rights page for the publisher (ACM), I as the author have rights to reuse the paper. The Authors' Rights can be found here: `https://authors.acm.org/author-resources/author-rights`. The DOI is yet to come as it was not published yet.

When the leader sends a request to the follower, it will append the index to a sliding window for that follower in order. Upon response from the follower, the leader frees all the requests that are no longer outbound. Second, the progress object also contains a `matched` variable that updates based on the response for an `AppendEntries` message. If the index in that response is outdated, the update on this variable will not succeed. Third, there is also a `next_index` variable that represents the index that is transmitted to the follower. The leader updates this variable primarily when it either transmits an `AppendEntries` message or receives an `AppendEntries` response.

Each field has an effect on the leader's decision to retransmit a previous `AppendEntries` message. The leader checks `Inflights` and `matched` when deciding whether to transmit another `AppendEntries` message after it receives a response from the follower (background task). First, when the leader receives a response to a previous `AppendEntries` message, it will check whether the progress object is paused. The paused condition can differ based on the state of the progress object. While the leader will pause the progress object when it transmits the next `AppendEntries` message if the object is in the `Probe` state, the leader will check whether the `Inflights` sliding window is full when it is in the `Replicate` state. Therefore, `Inflights` has a direct effect on the transmission of new messages. Second, the leader will check the `matched` variable against its own progress when it receives a heartbeat response. If the `matched` is less than the leader's progress, then the leader will send an `AppendEntries` message. While these fields cause the transmission of an `AppendEntries` message, a lack of updates in `next_index` will ensure that the transmission corresponds to a retransmission of a previous message.

The chance that the progress object is paused is higher when there is a fail-slow follower. In the `Replicate` state, `Inflights` is more likely to be full. Because the follower is slow, the time between adding and removing will be delayed. As a result, more messages will remain in the sliding window as new client requests push messages. Once the progress object reaches the `Probe` state, the likelihood of reaching the paused status is not significantly higher with a fail-slow follower. However, the leader will move the progress objects to this state more often with a fail-slow follower. The transition from the `Replicate` to the `Probe` state can occur when the leader receives a `MsgUnreachable`. The number of times this occurs is significantly higher with a fail-slow follower.

Furthermore, the fail-slow follower will delay the updates for the `matched` and the `next_index` variables. When the progress object is paused, the leader will skip sending requests to that follower. Because the leader will skip transmission of these messages, the `next_index` variable updates slowly. Moreover, responses received on those connections will have outdated indices that do not correspond to the latest client requests. Therefore, these

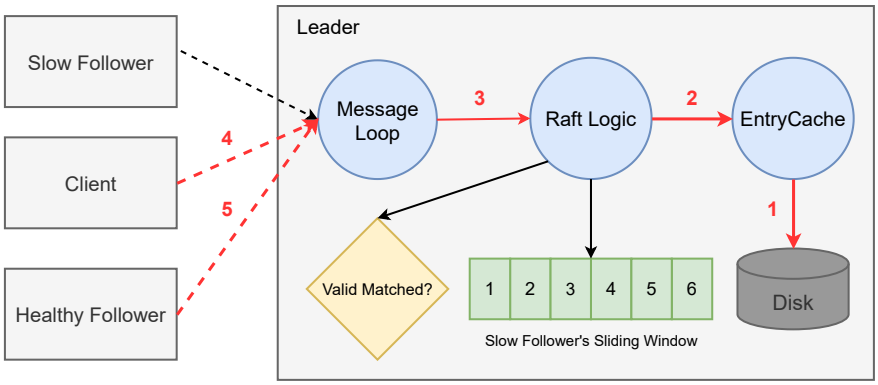


Figure 3.1: Slowness Propagation in TiDB

variables will not update and will cause subsequent retries on that connection.

We admit that there is some knowledge of the root cause that is missing. When the leader transmits a message to the follower after transitioning from the `Probe` state to the `Replicate` state, it will update the `next_index` to a higher value. In some cases, the `next_index` will decrease back to a smaller value, and the leader will retransmit an earlier `AppendEntries` message. We are investigating the reason behind this decrease further to see whether there is another issue apart from the repeated background tasks.

Summary: TiDB can spawn an arbitrary amount of disk and network activity when there is a slow follower by sending the same `AppendEntries` message to that follower more than twice. TiDB maintains a progress object with three fields that become problematic when there is a fail-slow follower. These fields (`Inflights` sliding window, `matched`, and `next_index`) act differently with a fail-slow fault and cause the leader to resend the same `AppendEntries` more than twice to the slow follower.

Figure 3.2: Summary of §3.1.1

3.1.2 Symptoms

To illustrate the symptoms of the root cause, we slow down a single follower with CPU contention in a 3-node configuration. We choose 256 concurrent client threads as this allowed the problem to be more visible than 512 threads used to saturate the server. We run 5-minute trials with and without a fail-slow follower to compare the results. From the TiDB internal metrics, we discover that a cache to bypass disk operations can experience 500-2000 cache misses, many of which stem from retries to the fail-slow follower. The cache misses can also

block other requests from the client and the healthy follower, further affecting the system's performance.

Before diving into our results, we will first describe the storage engine of TiDB, Raftstore, which performs replication. TiDB divides this store into multiple regions for Multi-Raft. Each region is mapped to a single thread, but a thread may account for multiple regions. Since every disk I/O operation is synchronous, the leader will maintain a cache to store recent entries. Before sending an `AppendEntries` message to the follower, the leader will attempt to read from the EntryCache. Inside the `entries` function, a miss in the entry cache will trigger a disk operation, a range scan.

We present the slowness propagation of resending messages in Figure 3.1. The arrows represent the order of execution while the numbers represent how the slowness propagates. This image demonstrates a scenario in which a retransmission of an `AppendEntries` message occurs either by having a full sliding window or a `matched` variable below the leader's index. For the sake of simplicity, we do not include the case in which the progress object is in the `Probe` state. The first point of propagation occurs because the Raft logic will call `entries` and could wait for an expensive and synchronous disk operation to finish. Since retries of previous `AppendEntries` messages involve older entries, the probability of a range scan is much higher. The propagation continues because the leader reads messages in a loop from the client and the followers. When one of these messages is related to the Raft logic, the leader will execute the logic synchronously, which will be slow when the EntryCache misses. This iteration will delay all the other important messages in the loop such as another client request or a response from a healthy follower.

During our experiments, we observe the effects of a fail-slow follower on the latency of `handle_msgs`, which is the message loop. The P99.9 and P99.99 tail latency of `handle_msgs` is 51.7% and 324% higher respectively. Most of the latency of `handle_msgs` also stems from the performance of `fetch_entries_to`. Approximately 78.19% of the latency for `handle_msgs` is the latency of `fetch_entries_to` in one of our trials. However, not all of the latency stems from the conditions that we reported earlier. In fact, cache misses from `AppendResponses` comprised 2.62% of the latency of `handle_msgs` while `HeartbeatResponses` contributed 6.85%. Although we honestly report these small contributions, we describe the potential of this root cause pattern to cause even more harm to the system in a later section.

Summary: An `AppendEntries` message to the follower might trigger an EntryCache miss. A retransmission of the request involves an older entry that might not be in the cache currently and can lead to an EntryCache miss and disk operations. Since messages are processed in a loop, the miss can affect the processing of other requests in the loop.

Figure 3.3: Summary of §3.1.2

3.1.3 Source Code Analysis

We present the code on the leader for sending an `AppendEntries` message to the follower and processing a response from the follower. We eliminate several branches and replace parts of the code that do not pertain to our problem with ellipsis. Furthermore, we utilize debug messages to show the effects of a slow follower on these code snippets.

Code Snippets In the first code snippet, we present a simplified code snippet of the sending logic in the figure below (Figure 3.4).

The code will call `maybe_send_append`, which sends an `AppendEntries` message to the follower under conditions.

```
if pr.is_paused() {
  return false;
}
if (...) {
  return false;
} else {
  m.entries = self.log.entries(pr.next_idx);
  match (term, ents) {
    (Ok(term), Ok(mut ents)) => {
      if self.state == Replicate {
        pr.optimistic_update(index);
        pr.ins.add(m.entries.last.index);
      }
    }
    ...
```

Figure 3.4: Simplified code of sending `AppendEntries`

```
    }
  }
  self.send(m);
```

Figure 3.4 (cont.)

First, the leader will check whether the progress object of the follower is paused. As described earlier, this may occur when the sliding window is full or the state has transitioned to the `Probe` state due to a report that the connection is unreachable. If the conditions are satisfied, the leader will skip sending that message to the follower. The implications of this discontinuation is that `AppendEntries` responses received from the follower will be even further behind and will not update the `matched` variable significantly. Furthermore, the code will not update the `next_idx` causing future background `AppendEntries` messages to be retries.

The second code snippet displays the logic for the Raft leader when it receives an `AppendEntries` response.

```
match m.get_msg_type() {
  ...
  MessageType::MsgAppendResponse => {
    old_paused = pr.is_paused(); // result of is_paused() depends on state
    if pr.maybe_update(m.index) {
      // frees messages in sliding window up to index
      pr.free_to(m.get_index());
    }
    if old_paused {
      maybe_send_append(m.from, pr);
    }
  }
  ...
}
```

Figure 3.5: Simplified code of handling `AppendEntries` response

For an `AppendEntries` response, the leader will check whether the progress is paused and save the result in a separate variable. Due to the slow updates of `matched`, the leader has a high probability of resending a previous `AppendEntries` message. It is also important to note that this check occurs before the leader frees messages from the sliding window. If the `AppendEntries` response has a higher index than the current `matched` variable, then the

leader will first free space in the sliding window. Doing this prevents `maybe_send_append` from skipping the retransmission of a previous `AppendEntries` message.

Similarly, we display the code for the leader when it receives a response to a heartbeat message.

```
match m.get_msg_type() {
  ...
  MessageType::MsgHeartbeatResponse => {
    // ins here represents the sliding window
    if pr.ins.full() {
      pr.ins.free_first_one();
    }
    if pr.matched < self.log.last_index {
      maybe_send_append(m.from, pr);
    }
  }
  ...
}
```

Figure 3.6: Simplified code of handling a heartbeat response

A response to a hearbeat message will result in a check of whether the `matched` variable has reached the current index of the Raft log on the leader. If it has not, then a previous `AppendEntries` message will be retransmitted to the follower. Again, the sliding window is updated so that the retransmission inside `maybe_send_append` can succeed.

Analysis First, we show that the sliding window is full more often with a fail-slow follower, which increases the chances of returning true in `is_paused`. We print messages when the index is added to and removed from the sliding window. Afterwards, we compute the difference in the timestamps for the same index. For experiments with no fail-slow faults, we use the connection to any follower with the largest difference. On the other hand, we use the connection with the smallest latency for the CPU contention experiments. Even with this unfair comparison, the experiments without a slow follower resulted in an average latency of 59 milliseconds while adding CPU contention led to an average latency of 5.481 seconds. This disparity demonstrates that requests are removed more slowly from the window, causing the paused condition to succeed more times.

We also confirm that the paused condition is indeed more likely to satisfy. There are two checks for whether the progress object is paused or not, both of which contribute to resending the same `AppendEntries` message. The first check is in Figure 3.4 where the

leader will skip sending an `AppendEntries` message. The second is in Figure 3.5 where the leader will store the result in `old_paused`. The experiments without a slow follower did not satisfy the paused condition a single time in any trial. Meanwhile, the experiment with a fail-slow fault satisfied the condition 888,203 and 1,560 times respectively.

The condition in Figure 3.6 succeeds similar numbers of times with and without slowness. This condition succeeds 3,564 times with slowness compared to 3,461 times without slowness. The difference between the number of times the condition is satisfied might seem trivial.

However, there is a qualitative difference between the messages sent from this condition with slowness and those without slowness. Sending the `AppendEntries` message with the same index twice occurs often in TiDB even without slowness. The unusual phenomenon is that a fail-slow follower causes the leader to resend the same message more than twice. For instance, we count the number of times the same message is being sent after it has been sent twice. The number is 2,448 times with a slow follower compared to 0 without one. The second condition is responsible for an average of 1,212 additional retries out of the 2,448. As a result, the heartbeat responses when there is slowness causes the system to stray away from the default behavior, which only sends the same message at most twice.

Summary: The duration between when a specific request is added to the sliding window and removed is significantly longer when there is a slow follower. As a result, the `is_paused` condition will be satisfied more often leading to more retries. The second condition between the `matched` variable of the follower and the leader's progress was satisfied similar numbers of times. However, this condition corresponded to the first retry without a fail-slow fault while the condition led to additional retries with a fault.

Figure 3.7: Summary of §3.1.3

3.1.4 Impact Analysis

The latency of `handle_msgs` discussed in §3.1.2 is actually a small percentage of the end-to-end latency of TiDB. While the P99 and P99.9 end-to-end tail latencies in TiDB are in the order of 100 milliseconds, the tail latencies of `handle_msgs` are only in the order of 10 milliseconds. The primary reason is that other parts of the leader contribute to the tail more. For instance, the P99.9 tail latency of a part inside `handle_raft_ready` can be approximately 106 milliseconds Considering that the pattern had little effect on `handle_msgs`, the overall impact is minimal.

Despite the impact on TiDB, the root cause can hypothetically have a drastic effect on other RSM systems. Since disk performance can affect the performance of `handle_msgs`, a disk-intensive database may experience significant disk contention from synchronous writes. In our experiments outside the scope of this paper, we have observed that disk contention on the leader can be detrimental in every metric.

Moreover, TiDB mitigates this problem by dividing the log entries into multiple regions. EntryCache misses in one region might not hinder the performance of the database in regard to other regions. When `handle_msgs` for one region is slowed down by EntryCache misses, other threads can continue to process requests for other regions. Dedicating a single thread mapped to every region would exacerbate the problem because it would block the progress of requests in more regions.

Finally, a disk with slower performance such as an HDD can further increase the impact of the root cause. For our experiments, we used an SSD that could have lessened the influence. To simulate the potential impact of slower disk performance, we added two print statements inside the second branch of `fetch_entries_to`, which increased the latency of `entries`. Afterwards, we observed the P99.9 latency of `handle_msgs` increased to 225 milliseconds. For every latency measurement of `handle_msgs` above the P99 latency, we deducted the latency of `fetch_entries_to`. The P99.9 latency of `handle_msgs` decreased to 52 milliseconds, showing that the root cause can potentially add 100 milliseconds to the tail.

While the end-to-end latency of TiDB is not heavily affected by this root cause, there is a significant theoretical impact. Background RPC's can spawn additional network and disk operations, which are nontrivial problems.

Summary: While the latency of `handle_msgs` mentioned in the symptoms is a small percentage of the end-to-end latency, the potential impact of a similar bug on another system is nontrivial. For instance, heavy disk activity could be impacted by disk contention and having a single thread for every region would worsen the effect on end-to-end performance.

Figure 3.8: Summary of §3.1.4

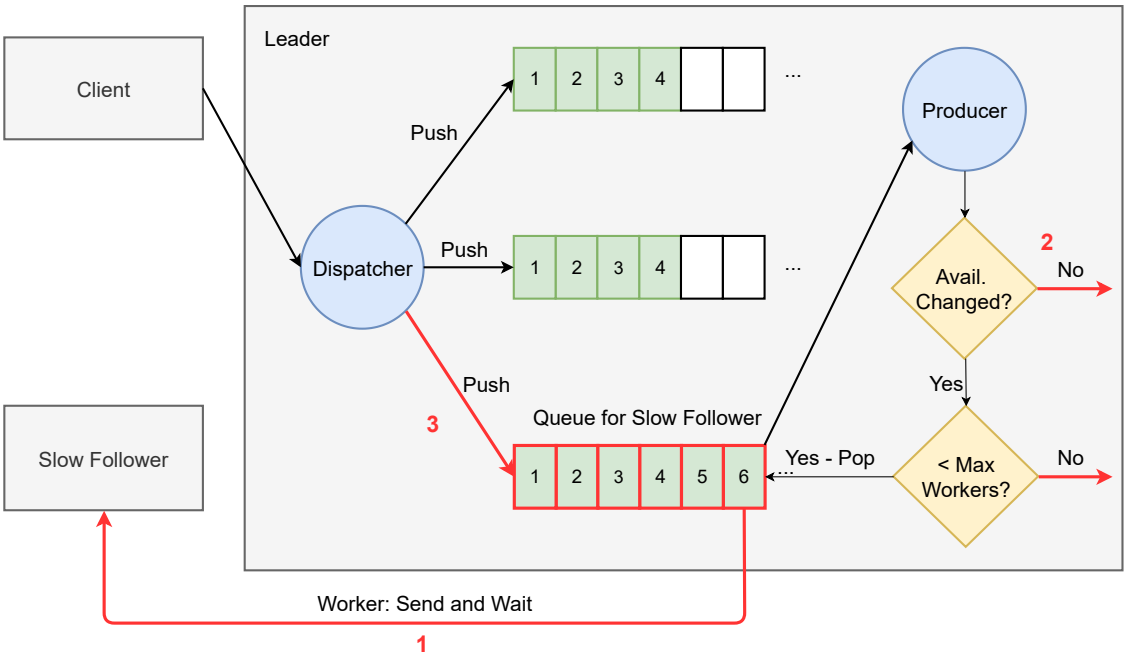


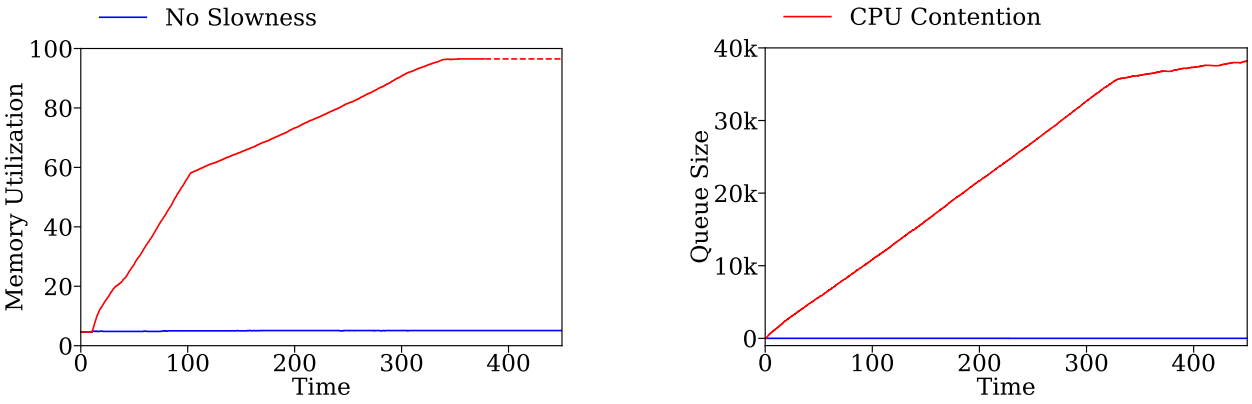
Figure 3.9: Slowness Propagation in RethinkDB

3.2 BACKLOG PROBLEM

3.2.1 Overall Description

Backlog problems are another pattern that causes a fail-slow fault to propagate. In RethinkDB, there is an unbounded buffer that results in higher memory utilization on the leader when there is a fail-slow follower. The size of the buffer can degrade performance towards the end of our experiments and even cause the leader to crash by running out of memory. The buffer grows because the fail-slow follower cannot respond with messages quickly enough for the leader to free the elements.

We show what happens when the leader dispatches a client request to a slow follower in Figure 3.9. At the initialization of the system, the leader defines multiple unbounded buffers for each replica. The dispatcher will first push a callback into buffers without any checks on the buffers' sizes when it receives a request. Each callback sends a request to the replica and waits for the response. To limit the number of coroutines, RethinkDB enforces several conditions before spawning new coroutines to execute the callbacks. In the context of the background queue, this means that it has either changed from non-empty to empty or vice versa. The second condition, which runs upon satisfying the first condition, checks whether the current number of worker coroutines is less than the maximum number of coroutines. If either condition fails, the producer will not pop from the queue.



(a) Memory Utilization

(b) Queue Size

Figure 3.10: Memory Utilization and Queue Size in RethinkDB

These conditions become an issue when there is a slow follower because the slowness can propagate to the leader. In Figure 3.9, we mark the propagation of slowness in red arrows from the follower to the leader along with the order in which it occurs. The current worker coroutines that execute the callback will wait for the response from the follower. The fail-slow fault in the follower delays the message and prolongs the callback. The leader cannot spawn new coroutines to accelerate popping from the queue due to the failed conditions.

Meanwhile, the leader does not check the size of the background write queue before pushing new requests from the client. This means that as the worker coroutines are not popping from the queue, new requests are repeatedly pushed to it. Consequently, the size of the unbounded queue continues to grow and eventually causes the leader to slow down and even crash due to out-of-memory issues.

Summary: RethinkDB contains background write queues for each server that store callbacks dependent on the performance of the followers. If there are a maximum number of coroutines that pop from the queue and execute the callbacks, then the leader will stop spawning them. A fail-slow follower will cause all these coroutines to wait, block the leader from creating coroutines, and increase the size of the queues.

Figure 3.11: Summary of §3.2.1

3.2.2 Symptoms

To highlight the symptoms of the root cause, we inject CPU contention in a follower and observe the effects on the queue size and memory. We add minimal log messages that confirm that our root cause occurs. We use the same number of clients as in Chapter 2 that saturates the leader and a 3-node configuration. Our experiment duration is 450 seconds that are enough to cause the leader to reach maximum memory utilization without crashing for most trials. Even with this rigid configuration, we could not completely control the leader from crashing. In one trial that we omit, the leader crashed immediately after 385 seconds. Changing the duration to 385 seconds would not be enough to underscore the impact of high memory utilization.

We present the changes in memory utilization and background write queue size over time in Figures 3.10a and 3.10b. We extract the maximum queue size of each follower in the trials and compute the median value across all the trials. We then plot the memory utilization corresponding to that trial by calling `top` every 0.1 seconds. Since Linux commands require sufficient memory to execute, errors will occur towards the end of trials. Therefore, we mark the end of these trials with a dashed line to demonstrate that the memory utilization cannot be captured.

The trends of the background queue size and memory clearly change once we inject CPU contention into a single follower. When there is no slow follower, the background write queue experiences little to no change. With a slow follower, the background write queue increases at a nearly linear rate. In fact, the maximum size of the background write queue was at least 25,532 with slowness while it only exceeded 100 for a single trial without slowness. The memory utilization shows a similar trend to the queue sizes.

Summary: We ran experiments to highlight the symptoms of the root cause. In the experiments without any slowness, the queue size and memory utilization do not change significantly. After slowing down a follower, we found that both of these metrics increase at a similar rate. In addition, the memory utilization at the end of the experiment is nontrivial as it approached 100%.

Figure 3.12: Summary of §3.2.2

3.2.3 Source Code Analysis

We analyze the code and print debugging message to further verify that the code is correctly associated with the root cause. First, we present code snippets from RethinkDB that include the conditions leading to the backlog problem. We simplify the code snippets to show the essential parts of the pattern. Then, we present the results from the debugging message that confirm the pattern and its causes. We recorded these numbers from the same trials as those analyzed in §3.2.2.

Code Snippets We provide the first code snippet inside `passive_producer.hpp`, which is where the first condition for popping from the unbounded queue is located.

```
void set_available(bool isEmpty) {
  if (isEmpty != empty) {
    empty = isEmpty;
    coro_pool->on_source_availability_changed();
  }
}
```

Figure 3.13: Simplified code of Condition 1

The argument to `set_available` is originally `bool a`, but it takes `!queue.empty` as input when it is called in the context of a background write queue. Therefore, we change the argument to `bool isEmpty` to place the code into a context relevant to our discussion. Whenever an element is pushed into the queue or popped from the queue, the leader will call `set_available`. Then, the leader will check whether the current state of the queue represented by `isEmpty` is different from the previous state represented by `empty`.

The behavior of the code in Figure 3.13 differs when we add a fail-slow fault. Without a fail-slow fault, `set_available` should call `on_source_availability_changed` afterwards in most cases. When the leader first receives a request and pushes to the queue, the queue should transition from empty to non-empty. The transition back to the empty state is normally quick enough that the next call to `set_available` calls `on_source_availability_changed`. However, a fail-slow follower will slow the callbacks down, causing less elements to be popped from the queue before the leader pushes more requests. This will cause future calls to `set_available` to fail more often than succeed unlike the case without a fail-slow fault.

The second code snippet displays `on_source_availability_changed` inside `coro_pool.hpp`, where the leader will pop from each background write queue and send the request to

the follower.

```
void on_source_availability_changed() {
  while (active_workers < max_workers) {
    ++active_workers;
    // run_worker will execute the callback
    // and pop inside a loop
    spawn_coro(std::bind(run_worker, this, queue->pop(), lock));
  }
}
```

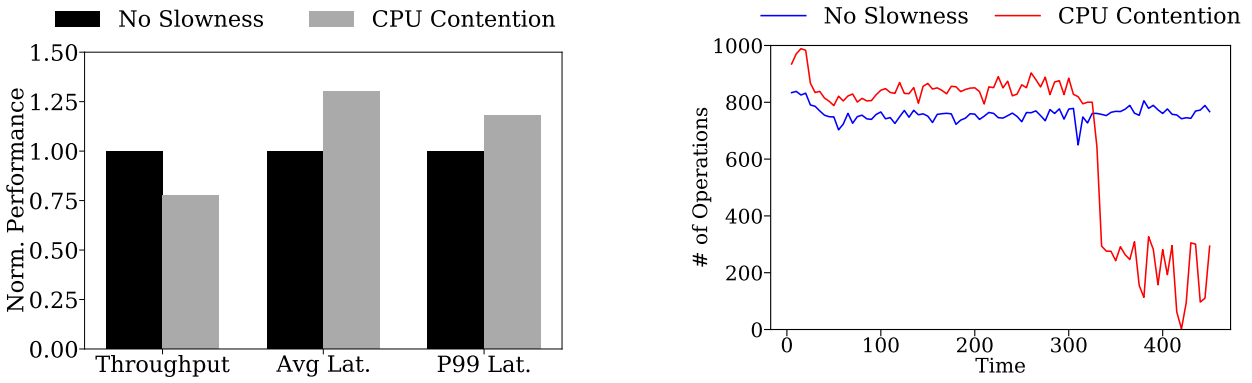
Figure 3.14: Simplified code of Condition 2

This step follows Figure 3.13 if condition 1 is satisfied. The code spawns worker coroutines that perform the task of popping and sending the request. Before the leader spawns these coroutines, it must check whether the current number of active workers is below the maximum workers. These workers will break once there are no objects left in the queue and decrease the `active_workers`.

The behavior of this code snippet also alters when we add a slow follower. When we have only healthy followers, the callbacks will run at normal speeds. The coroutines are more likely to finish before new elements are pushed into the queue. Once we add slowness to a follower, each callback is delayed by the follower as it waits for the response. Since callbacks are pushed faster than they are popped, the worker coroutines will rarely finish running. Meanwhile, the leader will continue to spawn new coroutines to process incoming client requests until the maximum is reached. Afterwards, the while loop will fail in calls to `on_source_availability_changed`.

Analysis We first record the overhead of each callback in the `background_write_queue` as this deters the other conditions from satisfying. In one experiment with CPU contention, we compare the overhead of all the callbacks for queues relating to the slow follower compared to the healthy follower. The average latency was 5.305 seconds for the slow follower while it was only 39.20 milliseconds for the healthy follower. As a result, the worker coroutines will not exit for an extended period of time and cause the conditions to fail.

We print out a debug message to verify that reasoning for condition 1 shown in Figure 3.13 if (1) the condition is not satisfied and (2) that check is for the background write queue. At the initialization phase, the leader will store ID's for each server that they connected with. We had each `coro_object` store the server ID. The message will print a message with the server ID upon a failure. Without fail-slow faults, the leader printed the message less than



(a) Performance Degradation

(b) # of Operations over Time

Figure 3.15: End-to-end Performance of RethinkDB

200 times for every follower. The experiments without a slow follower printed the message 360,298 times, all of which had the server ID of the slow follower.

We printed a different message for condition 2 that also included the server ID. The condition failed an average of 16 times for the CPU contention experiments while it failed an average of 6 times for the no slowness experiments. The disparity in the number of prints for the message is significant but smaller than condition 1. This is reasonable considering the order of slowness propagation. When the follower becomes slow, the leader will first spawn the maximum worker coroutines. Afterwards, the leader will push to the queue and call the first condition. The existing workers will not finish popping from the queue, causing the queue to remain non-empty. Since the first condition fails first, the second condition will rarely execute.

Summary: We presented the lines of code that represents the root cause of the increased memory utilization. We recorded the number of times the conditions in the code failed and saw an increase after slowing down a follower. For instance, the first condition failed less than 200 times across every trial without a slow follower but failed more than 300,000 times with a slow follower.

Figure 3.16: Summary of §3.2.3

3.2.4 Impact Analysis

We evaluate the impact of the buffer size and memory utilization on the end-to-end performance of RethinkDB. We plot the overall performance degradation in Figure 3.15a after

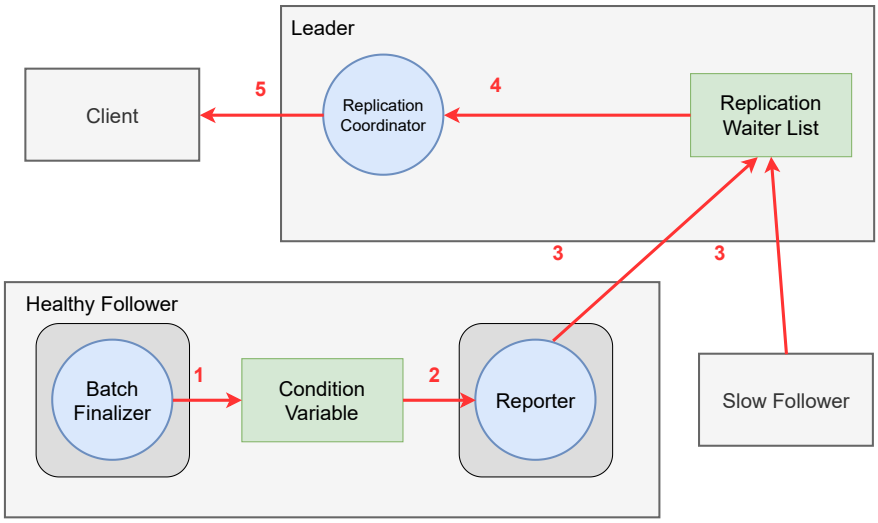


Figure 3.17: Flow of MongoDB Implementation

slowing down a follower. The throughput and average latency show degradation of greater than 20% while the P99 latency increases by approximately 17%. Therefore, every metric demonstrates significant performance degradation from a fail-slow follower.

Furthermore, we plot the changes in the number of operations over time in Figure 3.15b. We configure YCSB to print performance reports every 5 seconds. For most of the experiment, the performance disparity between the two experiments is minimal. In fact, the experiment with a fail-slow fault performs slightly better. Both experiments demonstrate a slight decrease in operations after the initial phase. After stabilizing, the difference is less than a few hundred operations for most reports.

However, after about 300 seconds into the experiment, there is a noticeable decline in performance for the CPU contention experiments. At the beginning of the experiment, the memory utilization has not reached critical levels as shown in Figure 3.10a. Therefore, the performance does not suffer from the increased memory utilization. However, the decrease in the number of operations occurs once the memory utilization exceeds 90%. For some reports in this period, the number of operations falls below 200, which is approximately a 75% degradation from the ∼800 operations that were common in other parts of the experiment and the experiments with healthy followers.

Summary: The end-to-end performance of RethinkDB suffers from the increased memory utilization. We show that every metric shows performance degradation after

Figure 3.18: Summary of §3.2.4

adding CPU contention on a follower. In addition, most of the performance degradation occurs at the end of the experiment when the memory utilization is at extreme levels.

Figure 3.18 (cont.)

3.3 TRANSIENT PERFORMANCE PROBLEM

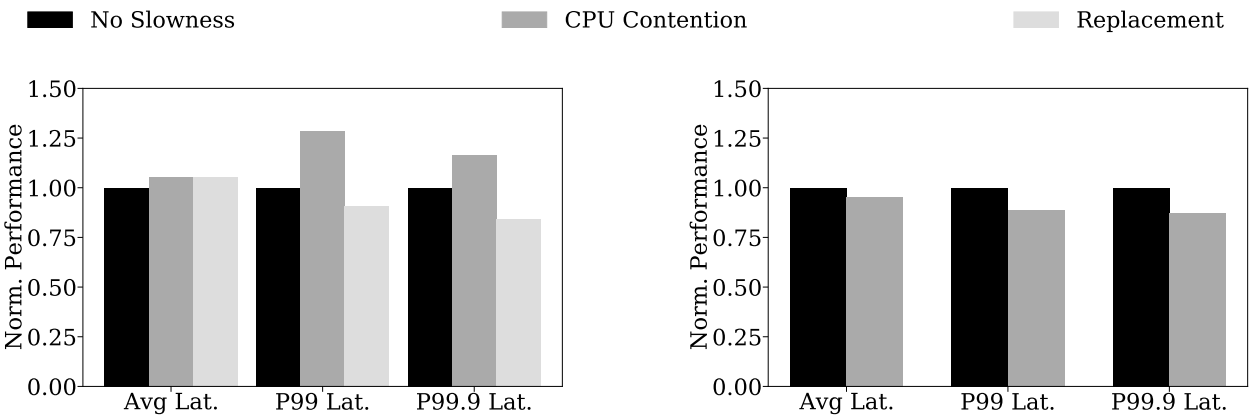
3.3.1 Overall Description

A fail-slow fault may render the system more susceptible to transient performance that can occur in various components of a distributed system. Spikes of several hundred milliseconds or even seconds are common in network or disk operations even in healthy followers. If we assume a 3-node configuration with the Raft protocol, a spike in one healthy follower will not impact performance as the other follower can satisfy the quorum. With a fail-slow follower, a spike in the healthy follower will result in a short period where both followers are slow. This problem is especially prevalent in MongoDB, which had multiple examples of this behavior.

Before diving into one of the issues, it is important to outline the procedure in which MongoDB performs replication since it has key differences from a traditional Raft protocol. The follower initially sends a `find` and `getMore` message to the leader after which the leader will send a batch for the follower to write. After these initial messages, the follower will no longer request for batches because the leader will proactively send batches to the follower. The follower processes the batch of log entries and sends a `replSetUpdatePosition` message to the leader.

There are two major instances in which the `replSetUpdatePosition` message is transmitted to the the leader. First, the follower will update the state of entries to applied and send the message, which we will abbreviate to `RSUP-applied`. Second, the follower will make the entries durable and send a `replSetUpdatePosition` message that we will call `RSUP-durable`. Because we enforce durable log entries and majority write concern in a 3-node configuration, the leader will wait until it receives an `RSUP-durable` message from a follower.

In Figure 3.17, we elucidate the key parts of the implementation of MongoDB pertaining to this particular pattern. We have a healthy follower and a slow follower in a 3-node configuration. When the client sends the request to the leader, the leader will wait for `RSUP-durable` messages by inserting a waiter into a replication waiter list. The leader will



(a) Performance of Request Execution on Leader

(b) Performance of `waitForJournalFlush`

Figure 3.19: Performance of MongoDB's Parts

Table 3.1: Effects of Journal Flush

Experiment	Number of Messages	Impact (ms)
No Slowness	523.8	-344.1
CPU Contention	87.42	63.42

unblock the waiter and send a response to the client when it has received an `RSUP-durable` message from a threshold of followers determined by the write concern. In a separate part of the implementation, the leader will send batches of client requests to the followers. The followers will call `waitForJournalFlush` in the batch finalizer after processing the batch of messages from the leader. They will then send an `RSUP-durable` message to the leader.

In this particular scenario, the call to `waitForJournalFlush` experiences a spike in latency and delays the `RSUP-durable` message the leader. We display the slowness propagation from `waitForJournalFlush` in red in Figure 3.17. First, both followers are slow in sending the `RSUP-durable` message at the same instance. The slowness of both followers will propagate to the leader because the waiters in the list depends on a message from either follower. Since the waiters are slow, the leader cannot fully execute the client query and the end-to-end latency of a client request will be affected.

Summary: MongoDB suffers from transient performance in the call to method

Figure 3.20: Summary of §3.3.1

`waitForJournalFlush`, which is imperative for sending a message to the leader that will unblock it. If we assume a quorum size of 3, a fail-slow follower will cause spikes in this method from the healthy follower that will propagate to the leader's performance. At this brief instance, both followers are experiencing a fail-slow fault.

Figure 3.20 (cont.)

3.3.2 Symptoms

To highlight the symptoms of `waitForJournalFlush` by running experiments on MongoDB using YCSB for 5 minutes. Like previous experiments, we compare experiments of the system without a slow follower in a 3-node configuration with experiments of it with a slow follower. Our workload size is 100 because it increased visibility of the problem compared to 320 clients used for saturation. We first explain the tail latency of `waitForJournalFlush` and then explore its effects on the leader in both configurations.

In Figure 3.19, we display the P99 and P99.9 tail latency on the leader to replicate the client request and of the call to `waitForJournalFlush`. Figure 3.19a presents the entire execution time on the leader, which includes waiting for replication to finish on a majority of the followers. This duration is significantly longer when we add CPU contention to a follower. The P99 tail latency increases by 21.288% while the P99.9 tail latency increases by 22.613%. We present the same numbers for `waitForJournalFlush`. In every metric, the latency of `waitForJournalFlush` is slightly more detrimental with two healthy followers than with a follower experiencing CPU contention.

Although the tail latency of `waitForJournalFlush` is higher without a fail-slow fault, we demonstrate that this latency is more likely to propagate to the followers with a fail-slow fault. We first recorded the number of times that `waitForJournalFlush` exceeded 100 milliseconds. We also printed the durable timestamp after the follower finishes running `waitForJournalFlush` and updates the timestamps. We then defined the interval for each `waitForJournalFlush` to be within the current durable timestamp and the durable timestamp after the previous `waitForJournalFlush` call. Afterwards, we measured the latency of executing the client request on the leader for each message in the interval. The difference between the latency of executing the client request and that of `waitForJournalFlush` is represented as "Impact" in Table 3.1. In addition, we also present the number of client requests for this interval.

When there is a fail-slow fault, the latency of `waitForJournalFlush` propagates to the

latency of executing a request on the leader. In the experiments without slowness, the latency of executing a client request is several hundred milliseconds lower than the latency of `waitForJournalFlush`. Meanwhile, the CPU contention experiments show that the execution time of a client request is larger than the latency of `waitForJournalFlush`. In most cases where `waitForJournalFlush` is slow in this configuration, the latency on the leader will be at least the latency of `waitForJournalFlush`.

This pattern can also affect the throughput as during this period of slowness in `waitForJournalFlush`, the leader can only process a limited number of requests with a fail-slow fault. In the experiments without a slow follower, the number of messages in this interval were significantly larger than 100. In the time that one follower is experiencing transient performance from `waitForJournalFlush`, the other follower can send multiple `RSUP-durable` messages to process new client requests. On the other hand, the experiments with a slow follower did not have a single `RSUP-durable` message that had an interval greater than 100 requests. Each `RSUP-durable` message can unblock up to a maximum of 100 client requests since the experiment has 100 clients. During this short period, neither follower will send `RSUP-durable` messages to the leader. This will cause the leader to send the responses to the client at a slower pace, severely limiting the number of client requests.

For each follower, 50.5% of the intervals for the `RSUP-durable` messages sent from a slow `waitForJournalFlush` led to a latency greater than 100 milliseconds on the client. At first glance, half of the cases seems to be problematic as both followers are healthy and can contribute a slow `waitForJournalFlush` that propagates to more requests.

However, most of the cases of propagation occurred when slowness was detected in `waitForJournalFlush` for both followers simultaneously. In fact, more than 85% of the cases in which the latency of `waitForJournalFlush` led to a high execution time on the leader were due to simultaneous slowness. In this scenario, the transient performance of both followers will propagate to the leader. This is also consistent with the results in Table 3.1 that showed the minimal effects of a slow `waitForJournalFlush` when there are two healthy followers. It is important to note that `waitForJournalFlush` is not the only cause of transient performance we have observed in MongoDB. Other transient performance issues could have coincided with slowness of `waitForJournalFlush` to account for the other 15% of the cases.

On the other hand, every case in which the latency of `waitForJournalFlush` was above 100 milliseconds coincided with a latency on the leader above 100 milliseconds. As a result, most of the latency values increased slightly between the `waitForJournalFlush` call and the execution time on the leader. This phenomenon is also reflected in Table 3.1.

We also demonstrate the impact of `waitForJournalFlush` on the tail latency of executing

a client request. First, we collected the execution time of a client request on the leader that corresponded with a `waitForJournalFlush` above 100 milliseconds for each trial. Then, we randomly sampled from this pool of latency numbers and replaced the corresponding latency values in the experiments with a slow followers. We used the same trial number to perform the replacement. The purpose is to simulate what would have been the latency if the latency of `waitForJournalFlush` had the same effect on the leader as the experiments without slowness. After replacing the latency values, we recomputed the P99 and P99.9 tail latency. Our results in Figure 3.19a show that the P99 and P99.9 latency both improve significantly. In fact, the results of this simulation outperform the P99 and P99.9 latency without any slowness on the system. It is clear that the P99 and P99.9 tail latency are greatly influenced by the latency of `waitForJournalFlush`.

Summary: We show the effects of `waitForJournalFlush` with and without a slow follower on the leader's complete execution of the request. When there was a `waitForJournalFlush` that incurred more than 100 milliseconds, the latency of the leader's execution of the request was also above 100 milliseconds. After replacing every request corresponding to high latency of `waitForJournalFlush`, we noticed a signicant improvement in performance, demonstrating the effects of this problem.

Figure 3.21: Summary of §3.3.2

3.3.3 Source Code Analysis

We present the code for MongoDB from the follower and leader to demonstrate how the slowness can propagate from the follower to the leader. Like the other patterns, our code snippets are simplified to only show essential parts of the pattern. The first code snippet represents what occurs when the leader receives an `RSUP` message from the follower. The second code snippet displays the loop that will call `waitForJournalFlush` and report the `RSUP-durable` message to the leader. We do not perform any analysis with debugging messages here as most of the analysis is already in §3.3.2.

The following code snippet inside `replication_coordinator_impl.cpp` is a method on the leader that will iterate through the waiters in the replication waiter list and wake them if they are ready.

```
void _wakeReadyWaiters(WithLock lk, OpTime opTime) {
  for (auto waiter: replicationWaiterList) {
    if (_doneWaitingForReplication(opTime, writeConcern){
      waiter->emplace_value();
      replicationWaiterList.erase(waiter);
    }
  }
}
```

Figure 3.22: Simplified code of Leader

The code is part of the logic for processing an `RSUP` message that the leader receives from the follower. Each `RSUP` message will include the highest timestamps of log entries that reached the applied and durable states. The method `_doneWaitingForReplication` must verify the the durable timestamp from the `RSUP` message instead of the applied timestamp as a precondition to returning true and unblocking the watier.

The second code snippet inside `oplog_applier_impl.cpp` represents the part that the follower calls `waitForJournalFlush` and triggers other parts of the code to send the `RSUP-durable` message.

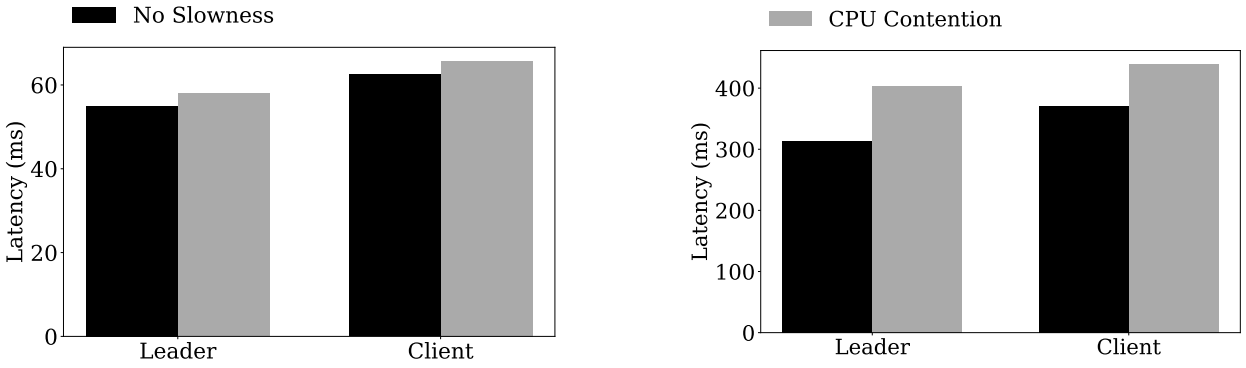
```
void ApplyBatchFinalizer::_run() {
  while(true) {
    waitForOplogApplier();
    JournalFlusher->waitForJournalFlush();
    // precondition for sending RSUP-durable
    _recordDurable(opTimeAndWallTime);
  }
}
```

Figure 3.23: Simplified code of Follower

First, the follower will wait for batches from the leader and apply the log entries. Afterwards, the follower might send a `RSUP` message to the leader, but the message will not pass the condition inside `_doneWaitingForReplication`. The follower needs to call `_recordDurable` to first update the timestamps for log entries that have reached the durable state. Then, the follower will send the `RSUP-durable` message to the leader to unblock the waiters. When there is a slow follower, the slowness propagates from the healthy follower to the leader through the latency of the `RSUP-durable` message.

Summary: Before sending a response to the client, the leader will wait until the data has been replicated to a single follower in a quorum of 3 replicas. The call to `_doneWaitingForReplication` will succeed only if the timestamp of the durable log entries has been updated by a follower. Since the follower will only update that timestamp after `waitForJournalFlush`, this call is on the critical path for processing a client request.

Figure 3.24: Summary of §3.3.3



(a) P99 Latency

(b) P99.9 Latency

Figure 3.25: Comparison between Latency on Leader and Client

3.3.4 Impact Analysis

We display the original values of latency that we normalized for Figure 3.19a and compare the values with the end-to-end latency of a client request. In both the P99 and P99.9 latency, most of the end-to-end latency is consumed by the latency of executing the query on the leader. We have already dissected the potential of `waitForJournalFlush` to affect the execution of requests. Therefore, most of the latency can propagate to the end-to-end tail latency when there is a slow follower.

CHAPTER 4: THE DEPFAST FRAMEWORK

We manifest our goals of resolving the aforementioned issues in the Dependably Fast Library(DepFast). [1] We summarize the components of DepFast as follows:

1. DepFast leverages a coroutine interface for synchronous code and to re-unite shredded code. DepFast achieves this by abstracting waiting points as events.

2. DepFast has an abstraction between utility as primarily basic event types and logic as advanced event types.

3. DepFast contains fail-slow tolerant events as part of the interface, which facilitates in preventing slowness propagation.

4.1 INTERFACE

The interface provided by DepFast is two-fold: (a) coroutine interface for new tasks and (b) event-based abstraction for waiting points.

Coroutines and events The coroutine interface aims to provide programmers an easy way to send and process requests. An event is a waiting point in which the programmer can code similarly to callbacks in an asynchronous model such as in an RPC. We present the following code example to demonstrate the use of the coroutine interface alongside the RPC component of DepFast:

```
Coroutine::Create([] () {
  for (auto rpc_proxy : servers) {
    auto entries = ...;
    // the next line bears possible slowness
    auto rpc_event = rpc_proxy.AppendEntries(entries);
    rpc_event.Wait(); // possible slowness
```

Figure 4.1: Fail-slow Intolerant Code

[1] This chapter reuses material from the following accepted paper: A. Yoo, Y. Wang, R. Sinha, S. Mu, and T. Xu, "Fail-slow fault tolerance needs programming support," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, Virtual Event, May 2021.

Authors are Andrew Yoo (author of this thesis), Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. According to the Authors' Rights page for the publisher (ACM), I as the author have rights to reuse the paper. The Authors' Rights can be found here: `https://authors.acm.org/author-resources/author-rights`. The DOI is yet to come as it was not published yet.

```
    if (rpc_event.timeout()) {
      ... // failure process
    } else {
      ... // process response
    }
  }
})
```

Figure 4.1 (cont.)

However, the above code does not meet our goals of providing an interface to combat fail-slow fault propagation. In one iteration of the loop, the RPC could be waiting for a response on a slow connection while the subsequent iterations cannot be run. Therefore, the slowness of one connection will propagate to the entire loop.

Quorum Events We present a new advanced event type `QuorumEvent` that helps developers meet all of our objectives. A `QuorumEvent` will wait for a customizable "quorum" in a collection of individual events. One popular use of this in the context of RSM-based prootocols is to wait for a majority. If we use a `QuorumEvent`, the coroutine is fail-slow tolerant when a minority of nodes are slowed down. Our own database DepFastDB exploits this event type to hinder slowness propagation from any slow node. To resolve the problem in the previous code, we present the following code that utilizes our `QuorumEvent`:

```
Coroutine::Create([] () {
  auto quorum_event = QuorumEvent();
  for (auto rpc_proxy : servers) {
    auto entries = ...;
    auto rpc_event = rpc_proxy.AppendEntries(entries);
    quorum_event.add(rpc_event);
    // no longer wait for any single event
  }
  // wait for a majority
  quorum_event.Wait(FLAG_MAJORITY);
})
```

Figure 4.2: Code with QuorumEvent

In the code above, a slow connection does not impede the loop as the RPC's are sent in parallel and the `QuorumEvent` is unblocked when a majority of RPCs are received. Consequently, a single slow connection should not increase the latency of the waiting event if we

assume independent and stable connections. The parts of the code with a `QuorumEvent` are labeled as fail-slow tolerant code. The idea behind our work is to advise programmers to write the logic with the `QuorumEvent` instead of waiting on basic RPC events.

4.2 MORE ON EVENTS

We categorize our events into two types: basic events and compound events. Basic events are primarily used for disk I/O and network operations. Simple waiting conditions such as waiting for a variable to be set is another basic event. Compound events combine multiple basic events.

`QuorumEvent` that we presented earlier is an example of a compound event. We also present the `AndEvent` and `OrEvent` types. The `AndEvent` is unblocked when every basic event inside of it is unblocked. The `OrEvent` is unblocked when a single basic event is unblocked.

We can also combine these events to represent more complex waiting conditions. Many algorithms requires not only waiting for a "majority-ok" but also waiting for a different condition such as "minority-plus-one-reject". As many programming models cannot represent this event in a simple yet precise manner, they replace it with a different condition such as "majority-reject". However, other conditions like "fast quorum" conditions are even more complex to represent [23, 24, 25].

4.3 RUNTIME

A runtime instance in DepFast is comprised of four major parts: coroutines, events, scheduler, and I/O helper threads. Coroutines are our unit of user tasks while events represent the waiting points. A scheduler in every runtime instance is responsible for pausing and resuming coroutines. I/O helper threads are background tasks that execute synchronous I/O such as flushing all disk writes to disk using `fsync`.

4.3.1 Runtime Verification

With the support of our events, runtime verification and tracing become simplified to allow debugging for slowness propagation. We can discover implementation bugs and account for protocol design optimizations between fail-slow fault tolerance and other principles of distributed systems such as load balancing in chained replication [20].

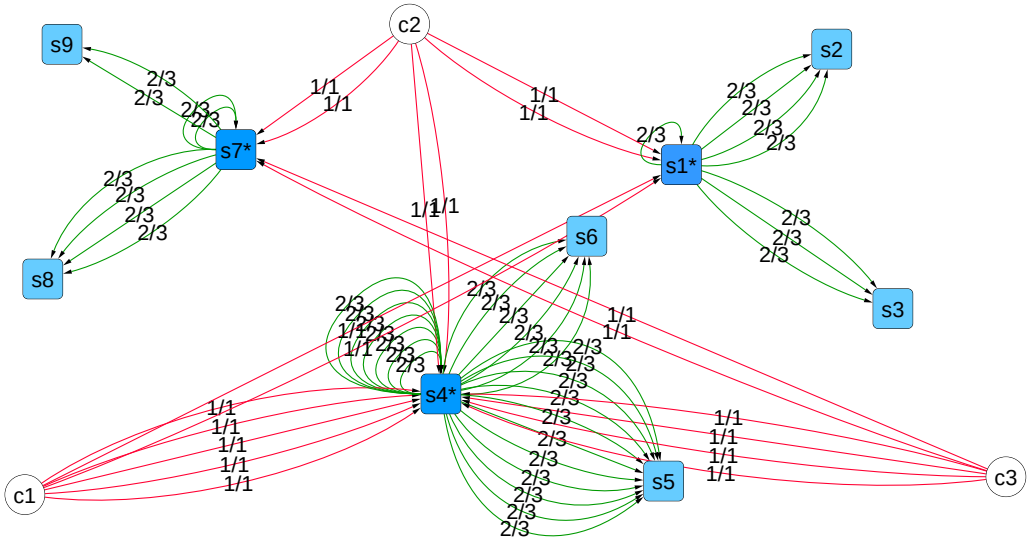


Figure 4.3: The slowness propagation graph. The labels on the edge represent the quorum of the event. "2/3" refers to the case where 2 responses are needed out of 3 RPCs; "1/1" refers to waiting on a single RPC.

Cooperation between multiple DepFast runtime instances facilitates runtime verification. Using events, DepFast connects coroutines on different server in a wait-for relationship. For instance, the caller will have a wait-for relationship on the callee for an `RpcEvent`. The following figure highlights an example of runtime analysis in DepFast.

We present a *slowness propagation graph* (SPG) that DepFast creates at runtime. We can utilize this graph to analyze the propagation from a fail-slow fault. In the example, we have a distributed database, DepFastDB, that has three shards each with their own quorum ({`s1`–`s3`}, {`s4`–`s6`}, and {`s7`–`s9`}). Each node in the graph represents a client (`c1` – `c3`) and a server (`s1` – `s9`). Each directed edge represents a wait-for relationship. A red colored edge represents a basic event that is not resilient to fail-slow fault while a green colored edge is a `QuorumEvent` that is resilient to a fail-slow fault. In each quorum, the SPG provides us confidence that there is not an event that is waiting on a single node. However, the clients are solely dependent on the leader, which demonstrates that the current system is not resilient to a fail-slow leader.

We also extend our runtime analysis to accommodate problems that we found in other databases.

Repeated Background Tasks Problem DepFast also provides information about the transitive dependency of each request at runtime to prevent the repeated background tasks problem. The transitive dependency of a request represents the dependency of the root of the request on the current recipient of the request. For instance, a client sends a request to the leader of a Raft-based distributed key value store and the leader broadcasts an `AppendEntries` message to the followers. By the transitive property, the client is also dependent on an `AppendEntries` acknowledgment from a majority of followers.

DepFast maintains a dependency ID that represents the transitive dependency of the request. The dependency ID is a pair with a string and an integer. DepFast requires that only the client should modify this dependency ID and every RPC request must contain a dependency ID. However, we make a single exception for heartbeat messages as they are common in distributed databases but do not originate from the client. Therefore, any server may modify the string to "`hb`" to represent a heartbeat message. Any other message should have a "`dep`" as the string and the actual ID as the integer in the pair.

If the dependency ID is not set or it is not a heartbeat message, then we know that the RPC request is a background task. Therefore, verification of background tasks can be done by checking the dependency ID of the request. When a configurable threshold of background tasks is reached or messages do not contain a valid dependency ID, the system will send out a warning to the programmer in the log messages. As a result, the repeated background tasks problem like the one in TiDB can be easily avoided with DepFast.

Backlog Problem DepFast also provides another interface for `QuorumEvents` that prevents backlog issues called `Finalize`. Each coroutine will maintain a list of `QuorumEvents` that have called `Wait`. For each `Wait` call, the programmer must also call a corresponding `Finalize` call with a timeout value and a flag. If the programmer does not call `Finalize` for every such `QuorumEvent` before the coroutine finishes, then DepFast will send a warning to the programmer that proper handling of dangling requests is necessary.

The `Finalize` interface attempts to process outbound requests for that `QuorumEvent` after a configurable number of calls. It will wait on a simple event type, `IntEvent`, that will only be satisfied once the leader has received a response from every follower. If the `IntEvent` times out, then the handler specified by the flag will be executed. The flag that we support is `FLAG_FREE` that will remove all the dangling requests from the buffer. We achieve this by pushing an IP address to a queue from which another thread will read and free the dangling requests for that queue. In this way, the Raft logic is not hindered by the overhead of freeing multiple dangling requests. This is only one example of a strategy that a programmer could implement to fix the problem.

The `Finalize` interface and our fix do not provide any mechanism for the follower to catch up with the leader. The runtime verification portion of DepFast is designed to facilitate programmers in avoiding the patterns that we observed in widely used databases. Therefore, the programmer can implement their own strategy to synchronize the slow follower.

Transient Performance Problem Our `QuorumEvent` interface can optionally track the history and latencies of previous `QuorumEvent`s to warn developers of the transient performance problem. To track the history, the developers can call `recordHistory` after the event is created and `updateHistory` when event is triggered. The history is a store with the keys as every unique `QuorumEvent` based on the recipients of the RPC's and the values as number of times each recipient triggered the `QuorumEvent`. For instance, the leader could send an `AppendEntries` message to itself (`s1`) and two followers (`s2` and `s3`). The key in this instance will be ({`s1`, `s2`, `s3`}). If the response comes from `s1` and `s2` first, then we will increment the counter for those recipients. We also have another store that maintains the latency of each RPC for previous `QuorumEvent`s.

The transient performance problem propagates when a majority-wait event is triggered by a subset of the nodes an overwhelming number of times. In our example, we can expect `s1` to almost always be responsible for triggering the `QuorumEvent` as it is the message from the leader to itself. Furthermore, the other two nodes (`s2` and `s3`) should be close to equally likely to also contribute to triggering the event if we assume both nodes to be healthy. However, if `s2` is slow for instance, then most if not all of identical `QuorumEvents` will be triggered by only `s1` and `s3`. As a result, transient problems in either of these connection will propagate to the leader.

The history and the latency stores can provide information about susceptibility of the database to the transient performance problem. We can traverse through the history to verify that every node was active in triggering the `QuorumEvent`. We define an inactive node to be a node that has not contributed to triggering a `QuorumEvent` for a sustained period of time. If the difference between the total number of nodes and the required number of responses is equal to the number of inactive nodes, then the current system is susceptible to transient performance. This means that every `QuorumEvent` is triggered by the same nodes for that period. When this is the case, we send warnings to the user that also contain the median latency, P99 tail latency, and P99.9 tail latency of the active connections. The developers can compare the P99 or P99.9 tail latency with the latency of client requests to analyze the severity of the problem. The disparity between the average and tail latency values might not be significant enough to address in some cases.

4.4 DEPFASTDB

We implement our own fail-slow tolerant Raft implementation using DepFast. The major components of the Raft protocol [9], leader election and data replication, involve sending requests to the other nodes and waiting for a majority of responses. Incorporating our `QuorumEvent` into these components is straightforward. Using DepFast, a masters student with basic distributed system knowledge implemented the Raft protocol in C++ within two weeks.

Our study of other databases helped us identify key patterns and write fail-slow tolerant code. For instance, an earlier version of DepFastRaft had the same backlog problem as RethinkDB before we analyzed other databases. We have an unbounded buffer that grows uncontrollably as removing elements from the buffer depends on the receipt of messages from a fail-slow follower. Since memory utilization on our database is high, the database did not remain at critical memory utilization to affect performance. We did not search for a prevention mechanism for this problem until we analyzed RethinkDB. The insights motivated us to avoid the same mistakes in other implementations. Through our runtime verification, we can ensure that our implementation is fail-slow tolerant

We implement our own RSM-based distributed key-value database on top of DepFastRaft to demonstrate our effectiveness in tolerating fail-slow faults. Our disk, network, and event processing is all provided by DepFast.

CHAPTER 5: EVALUATION

5.1 FAIL-SLOW FAULT TOLERANCE

5.1.1 Experimental Setup

We evaluate the performance of DepFastDB using the same fail-slow fault injection modes in Table 2.1 that we utilize for widely used databases (Section 2.4). [1] We ran experiments on an earlier version of DepFastDB that did not have the complete runtime verification. We test the database on a YCSB-like workload that writes to 100K keys. We evaluate the performance under a high load that can lead to about 75% CPU utilization on the leader.

In addition to the three-node setup that we used for other databases, we also measure the performance of our database under a five-node setup (one leader and four followers). We inject a fail-slow fault in one follower in the three-node setup and simultaneous fail-slow faults in two followers in the five-node setup. As a result, only a minority of the followers is affected by our injections.

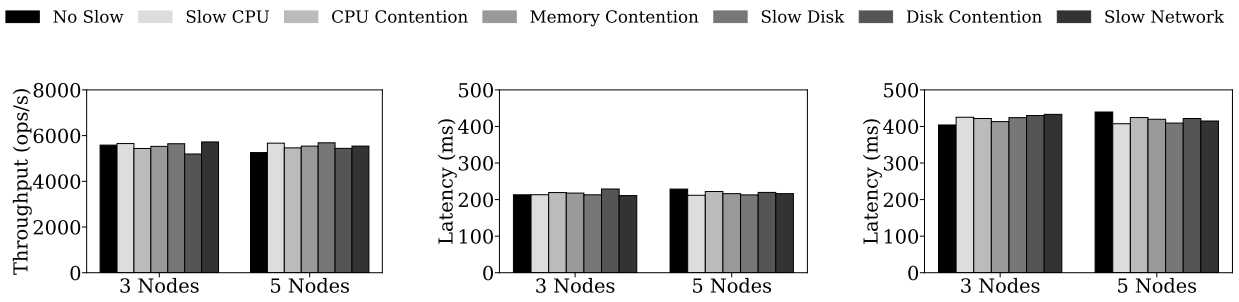
For a fair comparison, we use the same virtual machine setup as the widely used databases. We deploy the databases on the Azure cloud. Each virtual machine instance has 4 CPUs, 16GB RAM, and a 64GB SSD for data.

5.1.2 Results

Figure 5.1 presents the performance results under fail-slow faults on DepFastDB. In every metric, DepFastDB does not suffer any significant performance degradation from the injection unlike other RSM database implementations (Chapter 2). In fact, there were only 3 metrics out of 36 that showed performance degradation greater than 5%. The disparity between the performance of DepFastDB and the widely-used databases demonstrates that our database is more fail-slow fault tolerant. Furthermore, a 5-node configurration has minimal changes to the results.

[1] This chapter reuses material from the following accepted paper: A. Yoo, Y. Wang, R. Sinha, S. Mu, and T. Xu, "Fail-slow fault tolerance needs programming support," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, Virtual Event, May 2021.

Authors are Andrew Yoo (author of this thesis), Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. According to the Authors' Rights page for the publisher (ACM), I as the author have rights to reuse the paper. The Authors' Rights can be found here: `https://authors.acm.org/author-resources/author-rights`. The DOI is yet to come as it was not published yet.



(a) Throughput (b) Average Latency (c) P99 Latency

Figure 5.1: Performance of DepFastDB with a minority of fail-slow followers (different types).

5.1.3 Analysis

One of the main reasons to our tolerance to fail-slow faults is due to the DepFast framework. Instead of blocking on each RPC from the leader to the follower, the `QuorumEvent` interface allows us to transmit RPC's in parallel and wait for a majority. Therefore, we can be confident that slowness propagation does not occur in the broadcast of messages related to the Raft protocol.

This is not the only advantage of DepFastDB in comparison to other databases. Our runtime verification and analysis allows us to see a complete perspective on the system and identify slowness points easily. Using this information, we know that slowness of the followers should not propagate to the leader.

By using the runtime verification in our current version, we can be confident that there were no background tasks nor any significant transient performance in this early version. This version of DepFastDB is primarily missing the three runtime verification components based on the other databases. We did not add nor remove any background tasks and significant sources of transient performance from this version to our current version. The minimal number of warnings from our current version most likely means that background tasks and significant transient performance were not prevalent in the early version.

However, this early version is missing both the `Finalize` interface and our policy of freeing dangling requests. There is a chance that the results would have improved if we had integrated these into the early version.

5.2 RUNTIME VERIFICATION

We evaluate the effectiveness of DepFastRaft to detect the patterns discussed in Chapter 3 using DepFast. We inject the patterns by slightly modifying our implementation to

resemble these patterns in the databases that we evaluated on in §2.4. We create three variants of DepFastRaft and integrate them into DepFastDB. For each variant, we demonstrate the performance degradation of the pattern injection. We then analyze how well our implementation can identify the pattern and send a warning to the user.

Due to the limited functionality of DepFastRaft, we had to simulate the patterns discussed in Chapter 3 into our own to assess the effectiveness of the runtime verification. We do not have TiDB's mechanism for resending previous `AppendEntries` message upon receipt of an `AppendEntries` or heartbeat message inherently in DepFastDB. Furthermore, the transient performance problem is not as drastic in DepFastDB compared to MongoDB. Consequently, we created new fail-slow intolerant versions of DepFastRaft to resemble the patterns in TiDB and MongoDB. We call the variant with background tasks BG-Raft and the variant with greater transient performance TP-Raft for the sake of simplicity.

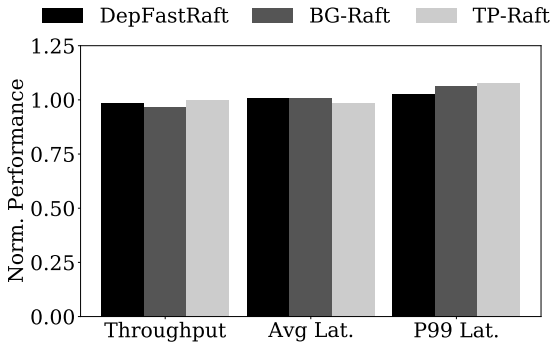
Simulating the behavior of RethinkDB did not require much modification to DepFastRaft because our implementation already has an unbounded buffer. Therefore, to simulate the backlog problem, we simply comment out every call to `Finalize` to ensure that dangling requests remain. In the experiment, the backlog problem became an issue in our system without `Finalize`. We refer to this variant as NF-Raft to denote that there are "no `Finalize`" calls.

After implementing the variants, we integrate our runtime verification to assess its effectiveness. Particularly, we present the number of warning messages that are printed in each scenario. We also test DepFastRaft on this runtime verification interface to expose any potential false positives.

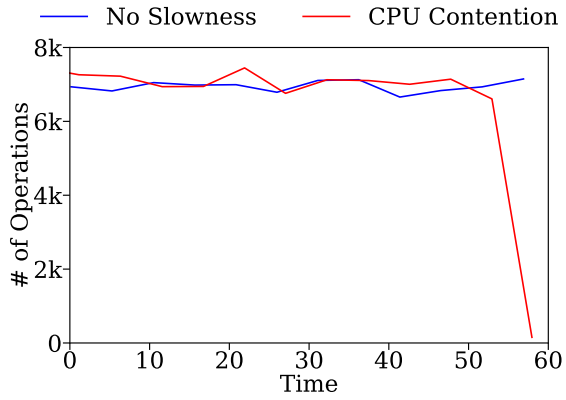
5.2.1 Experimental Setup

To evaluate the performance degradation of the injections, we run similar experiments to those performed in Chapter 3. We run CPU contention experiments using a YCSB-like performance benchmark that updates 100K keys. We choose 450 concurrent clients, which can drive the leader to about 75% CPU utilization Finally, our configuration has 3 nodes with one leader and two followers, and we will inject CPU contention to a single follower.

Backlog Problem We have to make modifications to our experimental setup to better visualize the performance degradation from the backlog problem. As observed in RethinkDB, the performance degradation from this problem is only visible when the database reaches critical memory utilization. It would be difficult to observe the impact of memory utilization when it is low.



(a) Performance Degradation of BG-Raft and TP-Raft



(b) Performance Degradation of NF-Raft

Figure 5.2: Performance of Fail-Slow Intolerant Variants

Table 5.1: Number of Warning Messages

Experiment	DepFastRaft	BG-Raft	TP-Raft	NF-Raft
No Slowness	0	0	0	21,155
CPU Contention	108	143	105	21,288

This becomes complicated in DepFastRaft because memory utilization is significantly higher than RethinkDB. In fact, DepFastRaft only experiences critical memory utilization for a small amount of time, approximately 5 seconds, under saturation. Choosing a correct duration that highlights this issue without crashing is a nearly impossible task. If the experiment runs a few seconds after the system reaches critical memory utilization, the system would crash. Meawhile, finishing the experiments earlier would not push the system to high enough memory utilization. Since every trial has a different throughput, the memory utilization across every trial is unpredictable.

We run experiments until a significant number of trials have reached critical memory utilization. We execute a `top` command at the end of the experiment duration and utilize the trial if the command fails. This is the same phenomenon that we experienced on RethinkDB towards the end of the experiment. For each of the trials, we plot the change in operations over time similar to Figure 3.15b.

5.2.2 Overall Results

We evaluate the performance degradation caused by each pattern that is injected into our database in Figure 5.2. The numbers in Figure 5.2a represent the normalized performance across the same metrics as those in §2.4 and §5.1.2.

We present the overall performance degradation after slowness injection on DepFastRaft in Figure 5.2a compared to BG-Raft and TP-Raft. The numbers represent the relative increase and decrease in every metric after slowing down a follower with CPU contention.

When there is no pattern, adding CPU contention does not affect the performance in our database. Across every metric, the largest performance degradation was 2.85% in the P99 tail latency.

BG-Raft and TP-Raft add more performance degradation in the P99 tail latency. BG-Raft's P99 tail latency increases by 6.42% after injecting CPU contention while TP-Raft's P99 tail latency increases by 7.78%. This is reasonable considering that MongoDB's transient performance problem also heavily affected the tail and our simulation of TiDB's background tasks problem spawned background tasks in intervals of several milliseconds.

In Figure 5.2b, we compare the two scenarios of the system with and without a fail-slow fault after commenting out every call to `Finalize`. We demonstrate the experiment that yielded the median of the maximum queue size, which is the same criterion for experiments in §3.2. For most of the experiment, both scenarios experience indistinguishable levels of throughput. When the memory utilization becomes critical as noted by the failed `top` command, the performance degradation becomes clear. The number of operations in the last statistics report before averaging the value over time was 769. This number on average across every trial is 12,141 across every experiment with CPU contention compared to 33,562 without slowness.

In Table 5.1, we display the number of warning messages in each variant both with and without slowness. For every variant with slowness, an ample number of warning messages are printed to inform the user of the pattern. Therefore, our runtime verification can successfully warn users when there exists slowness.

The only other interesting scenarios to discuss are the experiments of DepFastRaft with CPU contention and the experiments of NF-Raft without slowness. Our warning for transient performance problem is to warn the system that their system is susceptible to the pattern. CPU contention will cause the history inside our `QuorumEvent` to be dominated by a single follower regardless of the implementation of Raft. To assess whether to fix the transient problem or not, the user can look at the metrics that are printed in each warning. NF-Raft prints warnings based on whether the user is dealing with backlog rather than on the existence of backlog. The fact that the user is not calling `Finalize` is independent of whether there is a fail-slow fault in the system or not. As a result, both numbers are expected in DepFast.

Summary: Our results demonstrate that the patterns have a significant impact on the performance of our database. We also show that our runtime verification can accurately detect every pattern.

Figure 5.3: Summary of §5.2.2

5.2.3 Repeated Background Tasks Problem

Patterns To create additional background tasks, BG-Raft has a loop in the leader that will spawn background tasks based on messages received from heartbeat messages. The leader will attempt to send heartbeat messages to each follower every 5 milliseconds in the loop. When the follower receives the heartbeat message, it will respond with its current log index to the leader. For every heartbeat response, the leader will update a data structure call `matched` that represents the current progress of each follower. In every iteration of the loop, the leader will check the progress of the followers and resend the `AppendEntries` message if the `matched` variable is 10,000 entries behind the leader.

Our simulation replicates the pattern that TiDB exhibits: the leader is transmitting RPC's that does not stem from a client request. Like TiDB, BG-Raft will send the same `AppendEntries` message more than once based on a progress object. While TiDB resends an `AppendEntries` message in the event of receiving a message, BG-Raft transmits these messages based on the progress object alone. Despite the difference, we successfully achieve our goal of implementing a database that generates an excessive number of background tasks.

BG-Raft spawns more background RPC's when a follower experiences slowness compared to the scenario without slowness. In the trials, there were only 20 retries in the experiments without slowness while this number increased to 5,599 with slowness. Therefore, our simulation correctly simulates the behavior of spawning more background tasks in response to a fail-slow follower. These extra background RPC's also noticeably increased the end-to-end P99 latency in Figure 5.2a.

Summary: To simulate the behavior of TiDB, we implement the pattern of resending `AppendEntries` messages after receiving heartbeat responses on top of our database. We discovered that after injecting CPU contention, our database also retransmitted

Figure 5.4: Summary of Pattern Injection for BG-Raft

the same `AppendEntries` message multiple times. In addition, we demonstrate that the retransmission can increase P99 tail latency by 6.42 percent after we inject a fail-slow fault.

Figure 5.4 (cont.)

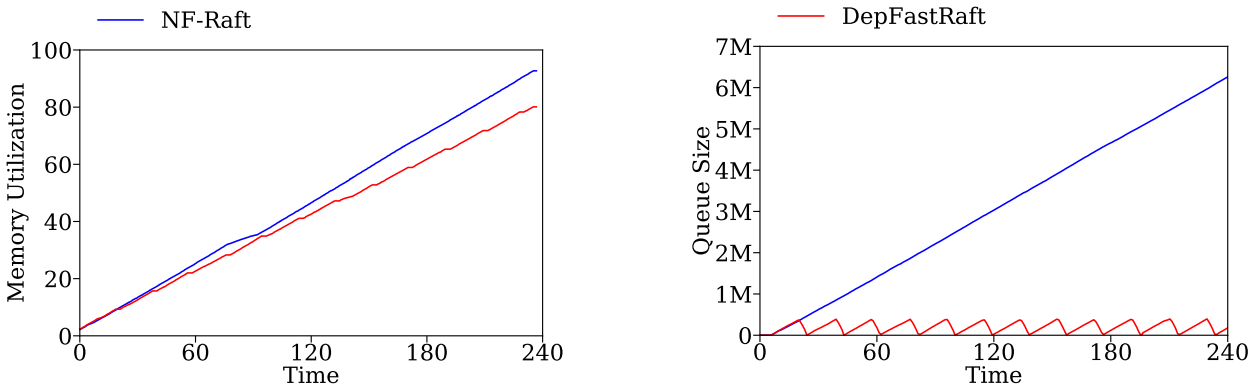
Results Our runtime verification can successfully detect the problems of BG-Raft and send warnings to the programmer. Every 5 seconds, the runtime verification interface determines whether to send a warning message if it has detected a message with an invalid or nonexistent dependency ID. In the trials with CPU contention, BG-Raft prints a total of 143 warning messages, 35 of which correspond to the background task problems. The other 108 messages stem from the runtime verification for transient performance since there is an inactive follower. Because our experiment duration is 180 minutes, we expect there to be approximately 35 or 36 checks for background tasks. Therefore, the interface is nearly perfect in detecting the pattern in BG-Raft.

The runtime verification for background tasks also does not result in false positives. In DepFastRaft, the warning message was not printed a single time as there are not many background tasks. In addition, BG-Raft does not print a message if the number of background tasks is small. In the no slowness experiments, the runtime verification message did not print a single time.

Summary: We print a warning message after checking periodically whether the number of background tasks has exceeded a threshold. The number of messages is expected based on the experiment duration.

Figure 5.5: Summary of Results for BG-Raft

Analysis DepFast can detect the background events created by the heartbeat messages because it tracks the transitive dependency of every message. With the pattern injection for BG-Raft, we have two additional types of messages from the original DepFastDB. The first message is the heartbeat message that does not need a transitive dependency. The second message is an older `AppendEntries` message triggered by the leader in response to the `matched` variable. This message is a background task that is not related directly to a client request and is not a heartbeat message.



(a) Memory Utilization (b) Queue Size

Figure 5.6: Memory Utilization and Queue Size with DepFastRaft

The second message, a retry of a previous `AppendEntries` message, never reaches the client that can set the dependency ID. Therefore, the leader will send the `AppendEntries` message to the follower with an invalid dependency ID. Our implementation detects this and maintains a counter for the number of background tasks. Once the number of retries reaches a threshold, the warning message is printed.

Summary: We exploit our tracking of dependency ID to detect whether a message originated from a client or not. Since the retransmission of older `AppendEntries` messages are started by the leader, DepFast can detect this and send warnings to the programmers.

Figure 5.7: Summary of Analysis for BG-Raft

5.2.4 Backlog Problem

Patterns NF-Raft simulates the backlog problems as the dangling requests in the unbounded buffer will remain. Whenever the leader sends a request to a follower, it will store the request into a `vector` corresponding to that follower. We will remove the object from the buffer and deallocate the dangling request when the leader receives a response from the follower server. In the case that the other server is slow, `Finalize` will help us deal with the dangling requests. Without `Finalize`, new client requests increase the size of the buffer as dangling requests are slowly being freed.

In Figure 5.6b, we present the size of the unbounded buffer with and without the `Finalize` interface. We remove the requirement that the memory has to reach critical utilization

for the experiments that we analyze for this section. We want to demonstrate that the symptoms of this problem occurs in every trial instead of selective ones. Again, we choose the median value of the maximum buffer sizes across every trial.

There is a clear discrepancy between the buffer sizes in DepFastRaft and NF-Raft. In DepFastRaft, the buffer size will grow to approximately 400,000 elements and drop immediately. The reason is that the `Finalize` calls will free all the dangling requests and reduce the size to 0. However, the buffer in NF-Raft will uncontrollably grow in a nearly linear fashion as we had previously observed in RethinkDB.

We also demonstrate the effects of the buffer on the memory utilization for the same trials in Figure 5.6a. The memory utilization will grow linearly even without the backlog issue because our database also stores data in memory. However, the unbounded buffer without the `Finalize` interface will increase the memory utilization significantly more. While the memory utilization only grows to 80.1% in DepFastRaft, removing the interface in NF-Raft will increase the utilization to 92.7%. As we described earlier, the critical memory utilization causes the throughput to plummet towards the end of the experiment.

Summary: We simulated the backlog problem by not calling `Finalize` at the end of every coroutine with `QuorumEvent`s. We demonstrate that NF-Raft leads to a sharp increase in queue size like in RethinkDB. Furthermore, the memory utilization is considerably higher in NF-Raft compared to DepFastRaft.

Figure 5.8: Summary of Pattern Injection for NF-Raft

Results Our runtime verification can reliably detect that the `Finalize` interface is not called. As stated earlier, we will check every coroutine after it has finished running a task whether it has finalized any `QuorumEvent`s that it waited on. Printing a warning in every task for a coroutine can significantly affect the performance of the server. Therefore, we output the message if the coroutine has not finalized every `QuorumEvent` in a 5-second period.

We warn the programmers often when they do not effectively handle the backlog problem. A message that the programmer should properly deal with backlog problems was printed an average of 21,155 times in NF-Raft without slowness and an average of 21,150 times with slowness. It is also important to note the higher number of warnings for transient performance (138 messages on average) was due to having to run longer trials to stress memory and injecting the fault several seconds into the experiment.

Compared to other variants, we print messages more often because we print a warning

message for each coroutine periodically. For other patterns, there exists one entity that determines whether to send the warning or not. Furthermore, since the message printing is independent of the existence of slowness, we believe that the 5 additional messages without a fault is trivial.

One worry that we want to dispel is the overhead of `Finalize`. Waiting for requests from a slow follower and freeing requests from a large buffer in a loop can both have significant overhead. However, the `Finalize` interface incurs almost no overhead. Compared to NF-Raft, DepFastRaft does not degrade performance by more than 5% in any metric.

Summary: Our runtime verification can reliably detect that no `Finalize` is called in NF-Raft. Furthermore, we also explain that `Finalize` incurs minimal overhead although freeing requests and waiting for a slow follower can potentially be time consuming.

Figure 5.9: Summary of Results for NF-Raft

Analysis We can deal with dangling requests because our event-based framework distinguishes logic from the utility and enables communication between them. Our Raft logic is implemented through `QuorumEvent`s that requires `Finalize` to deal with dangling requests. If the `Finalize` interface times out, then the `QuorumEvent`s will notify the utility that the logic is experiencing slowness. As a result, the utility can adjust by freeing all these dangling requests such that the slowness does not propagate to the buffer. In this manner, we can mitigate the impact of the backlog problem.

Our framework can send warnings since we incorporate `QuorumEvent`s into their own coroutine and recognize when the warning should be printed. In every coroutine, we maintain a data structure of every `QuorumEvent` that called `Wait`. Since every coroutine needs to finalize such events, we check this data structure at the end of the coroutine. If there was a coroutine that did not call `Finalize` in the last period of 5 seconds, then the coroutine code will print a warning. In DepFastRaft, we ensure that every coroutine calls `Finalize`, which avoids this warning from being printed. On the other hand, NF-Raft will print this warning often as every client request will proceed through the Raft logic without calling `Finalize`.

It is important to highlight that the `Finalize` interface have more disadvantages than advantages depending on the policy. For instance, we take extra caution when freeing the requests from the buffer. Freeing memory from a large buffer can be time consuming and can potentially block the critical path if implemented incorrectly. Therefore, we use a background

thread to minimize the overhead. In addition, waiting in every `Finalize` can have its own downsides. Defining a policy that minimizes the overhead is essential to reap the benefits of `Finalize`.

Summary: We can detect the lack of `Finalize` calls because we connect every coroutine with the `QuorumEvent`s that it waited on. Furthermore, defining the correct policy for `Finalize` is a nontrivial task that requires attention to detail.

Figure 5.10: Summary of Analysis for NF-Raft

5.2.5 Transient Performance Problem

Patterns We explain how we replicate the transient performance problem of MongoDB into our own database. DepFastRaft like most databases experiences inconsistent performance in network and disk operations. However, the transient performance problems are not as severe as MongoDB. As a result, we add artificial latency to each follower to simulate transient performance. We randomly generate a number whenever the follower processes an `AppendEntries` message and add 25 milliseconds if the number is divisible by 1,000. This adds artificial latency while mirroring the randomness of transient performance.

Adding artificial latency in TP-Raft had a significant impact on the leader's performance when there is a slow follower. We computed the latency of every `AppendEntries` message from the time that it was transmitted to the time of the callback. Without any slow follower, the round-trip P99 latency of an `AppendEntries` message was approximately 162 milliseconds. However, the latency is not visible on the latency of waiting on a `QuorumEvent` that depends on these messages. The P99 latency of waiting on `QuorumEvent` was just 73 seconds.

When we add CPU contention to a follower, the round-trip P99 latency of `AppendEntries` to the healthy follower is reflected in the P99 latency of waiting on a `QuorumEvent`. The P99 tail latency values of the `AppendEntries` request and waiting on a `QuorumEvent` are both 86 milliseconds. Although the P99 latency of each individual request is lower with a fail-slow fault, they have a more noticeable effect on the leader. The slowness of each message propagates to the leader as we observed in MongoDB.

Furthermore, the end-to-end P99 tail latency of every client request also suffers as a result of the artificial transient latency. The end-to-end P99 tail latency increased by 7.78% from 2.85%. This demonstrates the possible impact of the same pattern on our own system.

Table 5.2: Content of Warning Messages for Transient Performance

Variant	Median Latency(ms)	P99 Latency (ms)	P99.9 Latency (ms)
DepFastRaft	14.8	36.2	46.2
TP-Raft	18.7	84.0	95.5

Summary: We simulated the transient performance problem by adding artificial latency at random on the follower. We demonstrate that similarly to MongoDB, the transient performance does not propagate significantly to the leader without a fail-slow fault. However, when we add slowness to a follower, the artificial latency propagates to the leader.

Figure 5.11: Summary of Pattern Injection for TP-Raft

Results In every trial with CPU contention on a follower, we send multiple warnings that there is a possible transient performance problem. We use a 3-node configuration that enforces the leader to wait until the data is replicated on a majority of nodes. As a result, we should always print out a warning when exactly one follower is slow. When there is one fail-slow follower, transient performance of the other follower will propagate naturally to the leader. As we explained previously, both followers are experiencing slow performance for this short period.

In every variant except NF-Raft, the number of messages range from 105 to 108 for a 3-minute experiment duration. Meanwhile, NF-Raft printed the message 138 times due to the longer duration as we pointed out. We expect these numbers because three messages are printed whenever susceptibility to transient performance is detected. The first message states that the current system is susceptible to transient performance. The subsequent messages print the median and tail latencies of every active node from the moment that the `QuorumEvent` to when we notify the `QuorumEvent`. Therefore, we will output the first message followed by the latency values of the leader and healthy follower.

Although we print the warnings in every case with follower slowness, we output different information for TP-Raft that informs programmers of the severity. We show the metrics that the warning messages display for DepFastRaft and TP-Raft in Table 5.2. The extent to which transient performance occurs in DepFastRaft is small compared to TP-Raft. Since the end-to-end P99 latency only suffers by 2.85% in DepFastRaft, the programmers can choose to ignore this warning message. However, the tail latency in TP-Raft can have a more drastic

effect on the end-to-end latency. A problem such as the one in MongoDB where the tail latency of `waitForJournalFlush` can exceed a second would be highlighted by our warning message.

Summary: In every trial, our system successfully detects that it is susceptible to transient performance when there is a slow follower. Furthermore, the information printed when there was artificial latency informed us of a more severe transient performance problem.

Figure 5.12: Summary of Results for TP-Raft

Analysis We are able to detect the system's current susceptibility to transient performance on the follower because we exploit the `QuorumEvent`'s maintenance of the history and latencies of prior events. All of our `QuorumEvent`s from the `AppendEntries` phase of the Raft leader involves the leader node and the two follower nodes. Therefore, there will be an entry inside the history with this specific `QuorumEvent` as a key and a record of which nodes triggered the event as the value.

Periodically after a certain number of `QuorumEvent`s have been created, we traverse through the history to verify that the current system is not affected by transient performance. When we have two healthy followers, the history is divided between the leader and two followers such that one follower does not trigger the event significantly more than the other. The number of inactive nodes is 0, which is less than the requirement of one inactive node to print out the warning.

After we add CPU contention to one of the followers, the healthy node will help trigger the `QuorumEvent` while the slow node does not contribute. If the verification does not detect the dominance of the healthy node the first time, the next iterations will identify the problem. As a result, the single inactive node is sufficient to trigger the warning. The warning is not a false alarm because for the previous messages, the client requests are susceptible to a fail-slow fault in a follower. Since there is only one healthy follower, fluctuations in this follower will propagate to the leader as it was shown previously in our system and MongoDB.

Furthermore, the warning message itself is valuable because it demonstrates the impact of the transient performance on the end-to-end performance. In our experiments, we can analyze that the effects of the `QuorumEvent`s is significant by comparing the P99 tail latency from the warnings with the end-to-end performance. Without the artificial latency, the P99 tail latency was approximately 21 milliseconds higher than the median latency. Since the end-

to-end P99 latency was not affected, the developers can choose to ignore the message. While the information is printed even with the small transient performance, the number of warning messages is so minimal that the impact on performance is negligible. However, the P99 tail latency increased significantly as we added the artificial latency signaling the importance of the transient performance problem to the developers. In this scenario, the developers can make changes on other layers of the database to deal with transient performance.

Summary: The maintenance of history and latency work together to provide the information to the developers both that the problem is happening and the severity of the transient performance. Our periodic verification detects the inactive node and prints a warning. Meanwhile, the latency values are then utilized to evaluate the severity of the problem.

Figure 5.13: Summary of Analysis for TP-Raft

CHAPTER 6: RELATED WORK

6.1 REPLICATED STATE MACHINE SYSTEMS

The scope of our work is on RSM or replicated state machine [26] systems. [1] A replicated state provides fault-tolerant services to clients while providing an abstraction of a single entity. Normally, the client sends operations to the leader that then replicates it unto a majority of servers. A replicated state machine then guarantees through a consensus protocol [8, 9, 23, 24] that each server agrees on the same sequence of commands and every command is eventually committed.

Recent work in RSM systems most often describes that their system maintains fault-tolerance properties by design. Some papers explains their system's fault tolerance through previously proven protocols [2, 4, 27]. Other papers create their own protocol to fit the setting and prove that their system is fault-tolerant [25, 28]. While we have no doubt that the protocols themselves provide fault tolerance, our experience showed that implementations of such protocols can often be more problematic. In fact, none of the aforementioned work demonstrates that the implementations behave correctly in a fail-slow fault scenario. Our work aims to present an RSM system that not only follows these protocols but also verifies that the implementation itself is fail-slow-tolerant.

Our work is not the only comprehensive study that exposes unexpected behavior in RSM implementations [29, 30] Most of the studies on these implementations do not focus on such behavior in the context of fail-slow faults. Scott et al. [29] analyze an open-source implementation in akka-raft and expose behavior that occurs during fuzzing. Meanwhile, Fonseca et al. [30] discovered that bugs in verified RSM implementations were all related to the protocol implementations. However, none of the bugs exposed in these papers are directly related to fail-slow faults.

The backlog pattern was also reported in prior work on studies of ZooKeeper [14] and Verdi [30], but in the form of crashing behavior due to stack overflow. We reassure the existence of the same problem in a Raft implementation and show that backlog can lead to fail-slow behavior due to resource overuses. Note that Verdi is a formally verified implemen-

[1] This chapter reuses material from the following accepted paper: A. Yoo, Y. Wang, R. Sinha, S. Mu, and T. Xu, "Fail-slow fault tolerance needs programming support," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, Virtual Event, May 2021.

Authors are Andrew Yoo (author of this thesis), Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. According to the Authors' Rights page for the publisher (ACM), I as the author have rights to reuse the paper. The Authors' Rights can be found here: `https://authors.acm.org/author-resources/author-rights`. The DOI is yet to come as it was not published yet.

tation [31]—the current formal verification has not yet been able to deal with fail-slow fault tolerance properties. We also build an interface to deal with the root cause findings.

6.2 FAIL-SLOW FAULTS

Fail-slow faults are an active research topic, which includes both long slowdowns and transient faults with various root causes [12, 13, 14, 32, 33, 34, 35, 36, 37]. Work in this space also demonstrates the inability to tolerate such faults in state-of-the-art distributed systems [14, 15, 37]. One example as demonstrated by Do et al. [14] is *limplocks* where impact cascades such that the system slows down and cannot effectively fail-over to healthy components.

Root Causes Arpaci-Dusseau et al. [11] first introduced the "fail-stutter" (aka fail-slow) fault model for low and unexpected performance of different components. The work described different fail-slow faults caused by misconfigurations, resource contentions, and failures in the hardware itself. However, the work mainly focuses on presenting an overview of performance faults performed in a laboratory setting instead. Therefore, further work was needed to show the impact of these fail-slow faults in more detail.

Gunawi et al. [12] extended this work by performing a thorough study on reports of primarily fail-slow hardware faults. The study showed that every component, such as memory, SSD, and NICs, can have multiple root causes behind its failure. In addition to hardware failures that occurred internally, the paper also explains in detail external root causes such as power outages, temperature, and configurations.

In addition to extensive studies on hardware, other prior work has demonstrated the potential of software bugs [13, 14, 15] and misconfigurations [38, 39, 40] to result in a fail-slow fault. Some of the root causes caused by software include lock contention leading to additional tasks [15] and a slowdown from 1 Gbps to 1kbps stemming just from a network driver bug [14]. These prior investigations on software-related and hardware-related fail-slow faults provide more reason to explore fail-slow tolerant solutions such as our work.

Approaches Recent work focuses on leveraging monitoring, performance metrics, and measurements to detect and pinpoint the fail-slow faults [13, 33, 34, 35, 36, 37]. This reactive approach of detection and localization undeniably mitigates the problem, but a proactive approach can eliminate these occurrences. Our DepFast can prevent several impacts from fail-slow faults.

Prior work prioritized a proactive approach by designing new protocols that are fail-slow tolerant in their space. Mehdi et al. [41] built a causally consistent data store that is tolerant to slowdown cascades in which a single slow shard or component can dictate the performance of the entire system. The approach involved offloading the decision to commit to the clients and creating a weaker alternative to Parallel Snapshot Isolation (PSI) using timestamps. In a recent paper, Ngo et al. [42] developed the first consensus protocol that is tolerant to a single slowdown by design. The focus is to resolve the weakness of algorithms as they pertain to a fail-slow leader.

Our work is complementary as it focuses on the implementation level instead of the design level. The motivation behind our work is to ensure that the implementation can deliver on the principles of the algorithm. The aforementioned works [41, 42] and our work can be used in tandem to prevent slowness propagation. Furthermore, the scope of DepFast is not just a single distributed protocol but can be applicable to many similar algorithms.

CHAPTER 7: CONCLUSION

We exposed the fail-slow intolerance of three state-of-the-art RSM-based database systems. [1] None of the three systems could even tolerate a single slow follower. After an arduous process of searching for the root cause, we found that many of the problems stem from flawed implementation. We categorized the root causes into three patterns and performed a comprehensive analysis. TiDB spawns excessive background RPC's, RethinkDB has backlog problems involving an unbounded buffer, and MongoDB has transient performance issues that propagate to the leader.

We then extended DepFast, a library for writing fail-slow tolerant code, using insights that we have gathered from the databases. DepFast is a collection of interfaces that leverages coroutines and events to minimize slowness propagation. One of these interfaces is `QuorumEvents` where systems can wait on numerous events simultaneously until a majority finishes. In addition to this functionality, we help developers avoid all three patterns that we observed by sending warnings at runtime.

We finally implement DepFastRaft, a fail-slow tolerant implementation of Raft, and integrate it into a key-value store database (DepFastDB). We show that DepFastDB can effectively tolerate different fail-slow faults that affect other databases. In addition, DepFast can detect the patterns in other databases with near-perfect accuracy in our trials.

Future work for this project includes extending DepFast to other protocols. Moreover, we want to detect fail-slow faults that occur on the leader. The end goal is to have a library that can eliminate fail-slow intolerant code entirely. As we have demonstrated, implementing fail-slow tolerant code in distributed systems in general is a complicated task. DepFastRaft is one large step forward to achieving this goal.

A short version of the thesis has been accepted and will be presented at HotOS-XVIII [43].

[1] This chapter reuses material from the following accepted paper: A. Yoo, Y. Wang, R. Sinha, S. Mu, and T. Xu, "Fail-slow fault tolerance needs programming support," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, Virtual Event, May 2021.

Authors are Andrew Yoo (author of this thesis), Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. According to the Authors' Rights page for the publisher (ACM), I as the author have rights to reuse the paper. The Authors' Rights can be found here: `https://authors.acm.org/author-resources/author-rights`. The DOI is yet to come as it was not published yet.

REFERENCES

[1] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, "CockroachDB: The Resilient Geo-Distributed SQL Database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*, Portland, OR, USA, June 2020.

[2] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang, "TiDB: A Raft-based HTAP Database," in *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB'20)*, Tokyo, Japan, August 2020.

[3] M. Brooker, T. Chen, and F. Ping, "Millions of Tiny Databases," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, Santa Clara, CA, February 2020.

[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally-Distributed Database," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, USA, October 2012.

[5] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'10)*, Boston, MA, June 2010.

[6] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation (OSDI'06)*, Seattle, WA, USA, November 2006.

[7] M. Isard, "Autopilot: Automatic Data Center Management," *SIGOPS Operating System Review*, vol. 41, no. 2, p. 60–67, April 2007.

[8] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001.

[9] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*, June 2014.

[10] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[11] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "Fail-Stutter Fault Tolerance," in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS'17)*, May 2001.

[12] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, D. Srinivasan, B. Panda, A. Baptist, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li, "Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, Oakland, CA, USA, February 2018.

[13] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, "Gray Failure: The Achilles' Heel of Cloud-Scale Systems," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)*, Whistler, BC, Canada, May 2017.

[14] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi, "Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems," in *Proceedings of the 4th ACM Symposium on Cloud Computing (SOCC'13)*, October 2013.

[15] J. Li, Y. Chen, H. Liu, S. Lu, Y. Zhang, H. S. Gunawi, X. Gu, X. Lu, and D. Li, "PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems," in *Proceedings of the 39th ACM European Conference in Computer Systems (EuroSys'18)*, April 2018.

[16] "TiDB," https://pingcap.com/.

[17] "RethinkDB," https://rethinkdb.com/.

[18] "MongoDB," https://www.mongodb.com/.

[19] R. van Renesse and F. B. Schneider, "Chain Replication for Supporting High Throughput and Availability," in *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*, San Francisco , CA, USA, December 2004.

[20] MongoDB Documentation, "Manage Chained Replication," https://docs.mongodb.com/manual/tutorial/manage-chained-replication/.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC'10)*, Indianapolis, Indiana, USA, 2010.

[22] J. Ousterhout, "Why threads are a bad idea (for most purposes)," in *Presentation at the 1996 USENIX Annual Technical Conference*, September 1995.

[23] L. Lamport, "Fast Paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-112, July 2005.

[24] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'13)*, November 2013.

[25] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'15)*, October 2015.

[26] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, p. 299–319, December 1990.

[27] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, January 2011.

[28] J. Li, E. Michael, and D. R. K. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *Proceedings of the 26th Symposium on Operating Systems Principles*, New York, NY, USA, October 2017.

[29] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker, "Minimizing faulty executions of distributed systems," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, March 2016.

[30] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*, April 2017.

[31] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, June 2015.

[32] S. Jha, S. Cui, S. Banerjee, T. Xu, J. Enos, M. Showerman, Z. T. Kalbarczyk, and R. K. Iyer, "Live Forensics for HPC Systems: A Case Study on Distributed Storage Systems," in *Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC'20)*, Virtual Event, November 2020.

[33] B. Panda, D. Srinivasan, H. Ke, K. Gupta, V. Khot, and H. S. Gunawi, "IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services," in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, Renton, WA, July 2019.

[34] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, , M. Chintalapati, A. Krishnamurthy, and T. Anderson, "Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, USA, April 2018.

[35] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, "NetBouncer: Active Device and Link Failure Localization in Data Center Networks," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, Boston, MA, USA, February 2019.

[36] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred, "007: Democratically Finding the Cause of Packet Drops," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, April 2018.

[37] C. Lou, P. Huang, and S. Smith, "Understanding, Detecting and Localizing Partial Failures in Large System Software," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, Santa Clara, CA, February 2020.

[38] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. 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.

[39] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. 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, November 2016.

[40] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do Not Blame Users for Misconfigurations," in *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)*, Farmington, PA, November 2013.

[41] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I can't believe it's not causal! scalable causal consistency with no slowdown cascades," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, March 2017.

[42] K. Ngo, S. Sen, and W. Lloyd, "Tolerating Slowdowns in Replicated State Machines using Copilots," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*, November 2020.

[43] A. Yoo, Y. Wang, R. Sinha, S. Mu, and T. Xu, "Fail-slow fault tolerance needs programming support," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, Virtual Event, May 2021.