Fail-slow fault tolerance needs programming support

Andrew Yoo
University of Illinois
Urbana-Champaign, IL, USA
abyoo2@illinois.edu

Yuanli Wang
University of Minnesota
Twin Cities, MN, USA
wang8662@umn.edu

Ritesh Sinha
Stony Brook University
Stony Brook, NY, USA
rksinha@cs.stonybrook.edu

Shuai Mu
Stony Brook University
Stony Brook, NY, USA
shuai@cs.stonybrook.edu

Tianyin Xu
University of Illinois
Urbana-Champaign, IL, USA
tyxu@illinois.edu

ABSTRACT

The need for fail-slow fault tolerance in modern distributed systems is highlighted by the increasingly reported fail-slow hardware/software components that lead to poor performance system-wide. We argue that fail-slow fault tolerance not only needs new distributed protocol designs, but also desires programming support for implementing and verifying fail-slow fault-tolerant code. Our observation is that the inability of tolerating fail-slow faults in existing distributed systems is often rooted in the implementations and is difficult to understand and debug. We designed the Dependably Fast Library (DepFast) for implementing fail-slow tolerant distributed systems. DepFast provides expressive interfaces for taking control of possible fail-slow points in the program to prevent unexpected slowness propagation once and for all. We use DepFast to implement a distributed replicated state machine (RSM) and show that it can tolerate various types of fail-slow faults that affect existing RSM implementations.

KEYWORDS

Distributed systems, fail slow, fault tolerance, consensus

ACM Reference Format:
Andrew Yoo, Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. 2021. Fail-slow fault tolerance needs programming support. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31-June 2, 2021, Virtual Event*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465299

HotOS '21, May 31-June 2, 2021, Virtual Event

ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465299

1 INTRODUCTION

Fail-slow (a.k.a. fail-stutter) fault tolerance has long been desired by computer systems [4]. A fail-slow fault causes unexpected performance degradation of a hardware or a software component, without terminating or crashing the component (i.e., fail-stop) [4, 16]. A recent study shows that fail-slow faults can occur on all major hardware components, including CPU, memory, SSD, disk, and NICs [16]. Fail-slow faults can also be introduced in software components due to bugs and misconfigurations [15, 20, 27, 39]. Systems without fail-slow fault tolerance are prone to poor performance where just a single component fails slow [4].

Unfortunately, as shown in recent studies [4, 15, 16, 20, 23], many widely-deployed distributed systems cannot tolerate fail-slow faults. For example, Do et al. show that slowing down one node in five scale-out distributed systems can lead to cascading performance failures [15]. Gunawi et al. report that fail-slow faults constantly lead to chained events with cascading impacts across the cluster [16].

Recent efforts on combating fail-slow faults mainly focus on detecting performance cascading bugs [27] monitoring fail-slow runtime behavior [6, 19, 23, 34], and troubleshooting performance anomalies [3, 6, 29]. While those works provide remedies to the manifestation of fail-slow faults, a more fundamental direction is to build distributed systems that are inherently fail-slow fault tolerant.

We argue that fail-slow fault tolerance not only needs new distributed protocol designs, but also desires programming support for implementing and verifying fail-slow tolerant code. The argument is grounded by our observation that the inability of tolerating fail-slow faults in existing distributed systems is often rooted in the implementations and is difficult to understand and debug. In this paper, we based our discussion on RSM (Replicated State Machine) systems which are commonly designed for critical infrastructures [8, 11, 21, 22].

A RSM system consists of linearizable, fault-tolerant groups of distributed nodes coordinated using a consensus protocol (e.g., Raft [32]). In principle, a RSM system is supposed to

tolerate any minority of faulty nodes, as long as the majority are healthy. At the algorithm level, in a Raft-based system, a fail-slow follower should not have any user-visible impact by design. However, it is not the case for real-world RSM implementations. Our measurement shows that existing RSM implementations cannot consistently tolerate fail-slow faults on a minority of follower nodes (Section 2). To make matters worse, our experience shows that it is painful and time-consuming to debug the failures of fail-slow fault tolerance due to the challenges in understanding spaghetti implementations of request handling and replication procedures.

We showcase the benefits of programming support for fail-slow fault tolerance by designing the Dependably Fast Library (DepFast). DepFast aims to help developers implement distributed systems that faithfully guarantee the fail-slow fault-tolerance properties of the protocol algorithms (e.g., tolerating a minority of fail-slow followers in a Raft-based system). DepFast provides expressive interfaces for taking control of possible fail-slow points in the program to prevent unexpected slowness propagation once and for all. In the context of RSM systems, DepFast empowers the implementations not to wait on each event individually, but wait for a group of events collectively, until a majority finish. Furthermore, DepFast provides support for verifying and analyzing runtime behavior with regard to fail-slow fault tolerance (e.g., slowness propagation).

As a proof of concept, we use DepFast to build a fail-slow fault-tolerant Raft implementation (named DepFastRaft). We show that DepFastRaft can effectively tolerate various types of fail-slow faults which can affect other existing Raft-based systems. Specifically, the throughput, average latency, and P99 latency of DepFastRaft only fluctuate within 5% ranges with a minority of fail-slow followers.

2 CASE STUDY: RSMS

We discuss replicated state machines (RSMs) as a case study to show that implementations could break the fail-slow fault tolerance properties guaranteed by protocol algorithms. In principle, a RSM system can tolerate a minority of faulty nodes. We focus on fail-slow followers, instead of fail-slow leaders. In existing RSM designs (e.g., Raft [32] and Paxos [25]), a fail-slow leader would slow down the entire system, assuming no leader re-election. But, a minority of fail-slow followers should not have visible impact by design—a write can return after it is replicated to a majority of healthy nodes.

2.1 Measurement

We build a fail-slow fault injection tool. It injects different types of fail-slow faults (related to CPU, memory, SSD, and NIC) into the target systems and measures their impact on system performance. The fail-slow faults are simulated based on prior studies on fail-slow faults and represent common fail-slow modes [15, 16]. Table 1 describes those faults and the corresponding injection methods.

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 1: Simulated faults used for measuring fail-slow fault tolerance of existing RSM implementations (MongoDB, TiDB, and RethinkDB) and DepFastRaft.

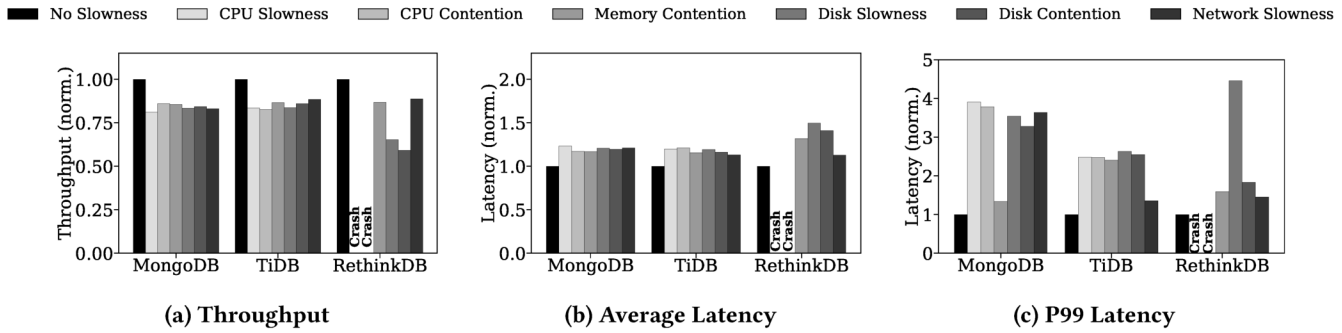
We run fail-slow fault injection testing on the RSM implementations of MongoDB, TiDB, and RethinkDB. For fair comparison, all the systems are set to use strong consistency when the consistency level can be configured (e.g., in MongoDB, `WriteConcern` is set to `majority` [35]). We turned off chained replication [37] which by design could propagate fail-slow faults [1]. For MongoDB and RethinkDB that have dedicated follower nodes, we inject the fault in randomly selected followers. TiDB uses a MultiRaft architecture where a node hosts both leaders and followers for different data ranges [18]. We configure a node to only host follower ranges and inject fail-slow faults on that node.

All the database systems are deployed on Azure cloud. Each node is deployed on a `Standard_D4s_v3` virtual machine instance with 4 CPUs, 16GB RAM, and 64GB SSD. We only evaluate three-node deployments of the three RSM implementations due to limited testing budgets.

We run Yahoo! Cloud Serving Benchmark (YCSB) [10] with and without the fail-slow faults (Table 1). The workload is a write workload that updates 500K records (we focus on writes because a write involves a majority of nodes). We run 256–1200 concurrent clients that drive the CPU utilization of the leader nodes to around 75% to represent a high load.

2.2 Results

Figure 1 shows the impact of a fail-slow follower in the three-node deployment of the three RSM implementations. We choose our baseline as the performance of each system without injecting fail-slow faults. We then normalize the performance of each system with specific fail-slow faults to



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

Figure 1: Performance of three RSM systems with a fail-slow follower (different types) under three-node setups.

its baseline performance. The normalization is necessary because different systems have different absolute performance.

Our measurement results show that existing RSM implementations cannot consistently tolerate fail-slow faults on a follower node. Specifically, a fail-slow follower can result in up to 17–41% decreases of system throughput decreases, 21–50% increases of average latency, and 1.6–3.46× increases of P99 tail latency across the three RSM implementations. In RethinkDB, fail-slow faults on CPUs crashed the leader.

Root causes. We observed the following root-cause patterns. First, fail-slow faults can be propagated due to synchronous wait behavior (the leader waits for the fail-slow follower). TiDB Raftstore uses a single thread for each data region. A fail-slow follower could force the leader to read old entries from the disk (those entries have been evicted from the in-memory EntryCache), thus blocking the whole thread during the disk I/O. Second, a fail-slow follower could lead to excessive backlogs at the leader side, which causes expensive processing and even resource exhaustion. RethinkDB maintains an unbounded buffer at the leader for outgoing writes—a slow follower can drive the leader to use excessive an amount of memory, or even run out of memory. Third, we observe that fail-slow followers have a significant impact on tail latency. With three-node cloud deployments, when one follower fails slow, transient performance issues on the other follower inevitably prolong the tail. Note that all the above root causes have been confirmed by the developers.

2.3 Discussion

In our experience, debugging failures of fail-slow fault tolerance is challenging and time consuming. It took two person-years to analyze the three RSM implementations of MongoDB, TiDB, and RethinkDB, regarding their behavior on fail-slow followers discussed in Section 2.1. The debugging process is mainly a binary search for the small fragments of code that caused the slowness based on printing timestamps. The process sounds easy, as we imagined it to be. However, in reality it is painful. The code often looks like spaghetti: the fragments that could affect a particular request are spread in different components and can be invoked at different events. Understanding where those code fragments are located and how they interact is non-trivial. Our experience working with developers of two of the three database systems shows that this is a challenging task, as it requires knowledge of all system components that a request could go through. These components are developped and maintained by different teams, which adds more difficulty in the diagnose.

Why spaghetti code? We do not have definitive answers. We present a few reasonings based on our own experiences and our discussions with the database developers. First, developers have been taught for a long time that non-blocking function calls with callbacks (a.k.a., asynchronous programming style) is preferred performance-wise over blocking function calls with threading (a.k.a., synchronous programming style) in building concurrent systems, especially in distributed systems [33]. There are many widely-used asynchronous event-driven libraries such as `libev` and `libuv`. In these libraries, developers write a "message loop", which parses the incoming messages and triggers relevant callbacks. Second, many distributed algorithms are written in a "upon receiving a particular message" style. For example, in Paxos papers, it is very common to read "when receiving enough Accept messages". If one already is familiar with the asynchronous message-loop programming style, it is very intuitive to translate the algorithm into the message loops. Overall, this will lead to the spaghetti code we discussed. Think about a Paxos system, for each request that goes through the 3 phases (Prepare/Accept/Commit) of Paxos, its code will at least be shredded into 3 callbacks. If this is a 5-replica system, the callbacks will be executed 15 times. If we include disk logging, there will be even more (at least double) callbacks. It does not take long before a developer loses track of how these callbacks could affect each other. It also painful (as we lived through) to monitor and manage the wait process.

Logic versus framework. We also observed that there is not a clear abstraction between the *logic* (e.g., the Raft logic) and the *framework* (e.g., code that implements RPC, disk flush for journaling, etc.). The problem of lacking the abstraction is two-folded. First, if we have a buggy fail-slow propagation, it is hard to know whether the bug is caused by the logic code or the framework code. A framework bug is usually easier for a systems programmer to identify and fix, compared with a logic bug. Therefore, it would be helpful if we can guarantee the logic does not have fail-slow issues. Second, lacking the abstraction also means lacking knowledge across the two parts. This means the framework code has to blindly execute the requests from the logic code and cannot perform any automatic optimizations to tolerate fail-slow failures, and has to push the burden back to the logic code. For example, the Raft logic broadcasts AppendEntries to all replicas and waits for a quorum of replies to proceed. In the current implementation, the Raft logic sends the same message to each replica and the framework code faithfully puts the message to the buffer of each replica. If one replica is slow, the connection will be slow and the buffer would keep increasing, leading to the backlog issue described in Section 2.1. If the framework is aware that this is a broadcast that can succeed with a quorum of replies, it can safely discard the messages for the slow connection.

Seeking for programming support. Our experience drives us to rethink the problem and seek for a more foundational solution that treats fail-slow fault tolerance as a first-class principle. We propose a framework that can solve the problem from the source. Specifically, this framework should: (1) re-unite the shredded code of asynchronous event-driven programming and make it easy to manage and monitor the waiting points on slow events, (2) provide a clean abstraction between the logic and framework code in order to support analysis and verification of the system behavior with regard to fail-slow fault tolerance, and (3) eliminate the cases where a fail-slow fault can be propagated to affect the implementation of other components. Overall, the goals of our framework are to: (1) isolate the fail-slow components and minimize their impact radius on their dependent components and thus to prevent fail-slow fault propagation, and (2) make it easy to understand and debug fail-slow behavior.

3 THE DEPFAST FRAMEWORK

To achieve the aforementioned goals, we built the Dependably Fast Library (or DepFast). To give a highlight of the DepFast design: (1) DepFast provides programmers with a coroutine interface to support synchronous programming and to avoid shredded code; it provides programmers with the event interfaces to wrap the waiting points, (2) DepFast implements utilities including networking, disk I/O, etc., and uses advanced event types as a clean abstraction between framework and logic code, and (3) DepFast supports fail-slow tolerant events, and these events can prevent the uncontrolled propagation of fail-slow faults.

It is worth pointing out that DepFast's design is based on the wisdoms in the discussions of asynchronous versus synchronous programming model [2, 7, 12, 14, 24]: we choose a synchronous programming model, but with lightweight and cooperative task scheduling (instead of a preemptive scheduling with heavy kernel threads).

In this section, we present how DepFast supports the implementations of replicated state machines.

3.1 Interface

The programming interface of DepFast mainly consists of two parts: (1) a coroutine interface for launching tasks and (2) an event abstraction for wrapping waiting points.

Coroutines and events. We provide a coroutine interface for a programmer to implement the logic of processing a user request. An event represents a wait point that will traditionally break the code into callbacks in an asynchronous model (e.g., an RPC). The following code snippet shows an example of Raft's algorithm using the coroutine interface and RPC interface 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
    if (rpc_event.timeout()) {
      ... // failure process
    } else {
      ... // process response
    }
  }
})
```

The example using coroutine can address the problem of spaghetti callbacks in the traditional asynchronous programming model. However, it is *not* fail-slow fault tolerant because a `rpc_event` could lead to slowness propagation—one slow RPC will slow down the entire loop, not mentioning that the RPCs are not sent in parallel as they should be.

Quorum events. To address the above issues, we introduce a new event type, `QuorumEvent`. As its name suggests, an `QuorumEvent` waits for a quorum or a collection of events (e.g., any majority). It allows the coroutine to tolerate fail-slow faults in any minority. The `QuorumEvent` is a key building block of DepFastRaft and prevents any single fail-slow component from straggling the entire system. With `QuorumEvent`, we can rewrite the previous example into the following code snippet:

```
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);
})
```

In the example, the RPCs are sent out in parallel, and the coroutine does not wait on any single RPC. In theory, any slow connection (or target server) should not affect the wait time, assuming the latency of waiting on each target is independent and stable. Therefore, we define code that only uses `QuorumEvent` and has no other waiting points as fail-slow fault-tolerant code. The principle of using the DepFast framework to write the logic code of a system is waiting on `QuorumEvent` as much as possible and avoid waiting on other types of singular-point events.

3.2 More on events

In general, DepFast provides two event types: basic events and compound events. Basic events are mostly for network and disk I/O events as well as other simple conditions such as waiting for a variable to be set certain value. Compound events are combinations of events.

`QuorumEvent` is a compound event and we have introduced it in Section 3.1. Other compound events in DepFast includes `AndEvent` and `OrEvent`. An `AndEvent` is triggered when all its subevents are triggered; an `OrEvent` is triggered when one of its subevents is triggered. Note that Events can be nested, e.g., an `AndEvent` can contain many `QuorumEvent`s as its subevents.

Nesting events can express complex waiting conditions. In many quorum-based systems, besides waiting for replies until "majority-ok", the algorithm either explicitly or implicitly states a different condition that is "minority-plus-one-reject". With a traditional programming model, this is hard to capture precisely and thus is often simplified to something easier to implement such as "majority-reject". However, in many cases, the waiting conditions could become complex to simplify, such as with "fast-quorum" based conditions [26, 30, 40]. With the `OrEvent` and `QuorurmEvent`, these conditions can be fairly easy to describe as follows:

```
QuorumEvent fast_ok = ...
QuorumEvent fast_reject = ...
OrEvent fastpath(fast_ok, fast_reject);
fastpath.Wait(/*timeout=*/1000); // 1000 ms
if (fast_ok.Ready()) {
  ... // process fast path
} else if (fast_reject.Ready() || fastpath.Timeout()) {
  QuourmEvent slow_ok = ...
  QuorumEvent slow_reject = ...
  OrEvent slowpath(slow_ok, slow_reject);
  slowpath.Wait(/*timeout=*/1000); // 1000 ms
  if (slow_ok.Ready()) {
    ... // proceed slow path
  } else if (slow_reject.Ready()) {
    ... // retry
  } else {
    ... // timeout: disconnect from group;
  }
}
```

3.3 Runtime

A DepFast runtime instance consists of four major components: coroutines, events, a scheduler, and I/O helper threads. As discussed in Section 3.1, coroutines are the units for executing user tasks and events mark the waiting points in the tasks. Each DepFast runtime instance has one scheduler to be in charge of suspending and resuming the execution of all coroutines. The I/O helper threads run in the background to deal with synchronous I/O events, e.g., the `fsync` calls that ensure that all disk writes have arrived at disks.

Runtime verification. Having events as trace points, DepFast supports runtime verification and trace analysis for fail-slow fault tolerance. This not only helps detect unexpected implementing bugs but can also be used to reason about design tradeoffs between fail-slow fault tolerance and other properties (e.g., load balancing in chained replications [1]).

Multiple DepFast runtime instances will work together for the tracing. DepFast links coroutines in different servers through the events. For example, a `RpcEvent` links the caller and the callee coroutines through a waiting-for relationship. We demonstrate an example of an analysis DepFast can provide. Based on linking the coroutines, DepFast can generate *slowness propagation graphs* (SPGs) at runtime. SPGs

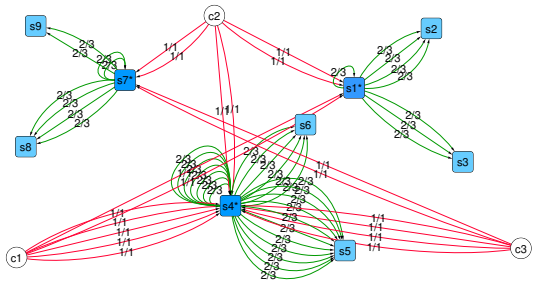
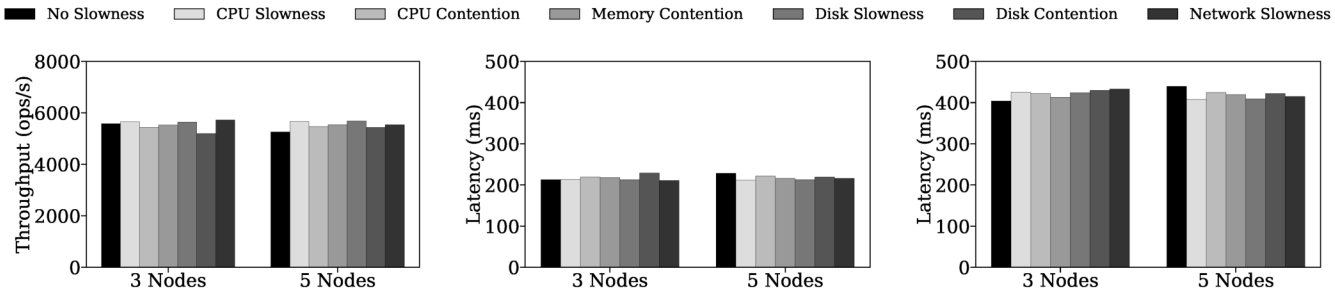


Figure 2: The slowness propagation graph. The labels on the edge represent the quorum of the event. "2/3" refers to the case where 2 responses are needed out of 3 RPCs; "1/1" refers to waiting on a single RPC.



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

Figure 3: Performance of DepFastRaft with a minority of fail-slow followers (different types of fail-slow faults).

can be used for analyzing fail-slow fault propagation. Figure 2 provides an example of SPG for DepFastRaft, which is deployed with three shards, i.e., three quorums ({`s1–s3`}, {`s4–s6`}, and {`s7–s9`}). The SPG is visualization at a node granularity by aggregating thousands of coroutines per node. Each vertex represents a node (`s1–s9`) or a client (`c1–c3`). Each edge is directed—the direction suggests the waiting-for relationship. Each edge is colored: a wait on a basic event (e.g., an `RpcEvent`) contributes to a red edge; a wait on a `QuorumEvent` contributes to a green edge. The SPG shows that there is no single-event wait in the interactions within each quorum (thanks to the uses of `QuorumEvent`). However, the clients wait for leader nodes—if a leader fails slow, the corresponding client will be affected.

We plan to extend the analysis to support more advanced and versatile analysis by integrating the probability models that consider transient fail-slow events.

3.4 DepFastRaft

To demonstrate the effectiveness of DepFast, we use DepFast to implement a Raft-based replicated key-value store, named DepFastRaft. Raft's protocol mainly consists of two parts, leader election and data replication [32]. Both follow the same pattern: a node broadcasts requests to other nodes and proceeds after it receives a quorum of acknowledgements. The pattern can be well expressed by `QuorumEvent`. Using DepFast, a Master student translated Raft pseudocode into stable C++ code in less than two weeks.

We evaluated DepFastRaft using the same fault injection based methodology for measuring existing RSM implementations (Section 2.1) and found that DepFastRaft is fail-slow fault tolerant, as shown in Figure 3. In all cases where a minority of follower(s) are slowed down, DepFastRaft's performance does not show performance drift over 5% in both latency and throughput. This is in contrast to existing RSM implementations (see Section 2.2). The base performance of DepFastRaft is at about 5K requests per second and outperforms other RSM implementations (so, the low drift is not because we have a smaller base performance).

4 RELATED WORK

Fail-slow faults have been actively studied recently, including both long-lived slowdowns and transient faults, with diverse root causes [5, 15, 16, 20, 28, 34, 36, 41]. Recent studies report that state-of-the-art distributed systems often cannot tolerant fail-slow faults [15, 27, 28]. For example, Do et al. show that fail-slow hardware can drive distributed systems into *limplocks*—cascading impacts where the entire system progresses slowly and is not capable of failing over to healthy components [15].

The main research efforts in addressing fail-slow faults focus on *detecting* and *localizing* fail-slow failures based on performance metrics [34], runtime monitoring [20, 28], and active measurements [5, 36, 41]. While detection and localization can greatly help resolve fail-slow faults, they do not eliminate the occurrence of the faults and are reactive to their manifestations. Our goal is to build distributed systems that tolerate various fail-slow faults in the first place.

Recently, Ngo et al. designed the first one-slowdown tolerant consensus protocol, Copilot [31]. It aims to address the algorithmic weakness of leader-based consensus protocols, i.e., the fail-slow leader (discussed in Section 2). Complementarily, DepFast focuses on system implementation. DepFast tries to guarantee that the implementation faithfully carries out the algorithm design properties. One can use DepFast to implement Copilot to avoid fail-slow fault tolerance at multiple levels. The design of DepFast is generic and is not specific to any distributed protocols.

In our experience, the interfaces DepFast provides can also help achieve a cleaner and more readable implementation of fault-tolerant (RSM) algorithms, which has its own merits because "fault-tolerant algorithms are notoriously hard to express correctly" [9]. On the direction of ensuring that the

algorithms are faithfully and correctly implemented, there are more rigorous approaches such as writing the algorithms with a specification model and later mechanically converting it to runnable code [9], or formal verifications [17, 38]. These works could also benefit from DepFast's interfaces.

5 SUMMARY AND FUTURE WORK

Our experience of developing and using DepFast is encouraging. As demonstrated by DepFastRaft, DepFast can help programmers implement fail-slow fault tolerant systems and be able to verify their runtime behavior.

We are working on enhancing DepFast for building different types of distributed systems other than RSMs, such as sharded data stores with distributed transaction protocols which also have complicated waiting conditions. We are also working on providing more observability through the event interface. We realize that the events in principle provide trace points needed by existing monitoring techniques [19] and the traces can be used for performance analysis [13, 29]. Therefore, we plan to implement failure detectors based on those trace points. Lastly, we will develop mitigation procedures specific to the detected failure modes. For instance, in DepFastRaft, if the leader is detected to fail-slow, a leader re-election can be triggered to turn the fail-slow leader into a fail-slow follower, which is well tolerated by DepFastRaft, as shown in Section 3.4.

ACKNOWLEDGMENTS

We thank David Daly, Indranil Gupta, Ger Hartnett, Jianjun Li, Darko Marinov, Madhusudan Parthasarathy, Liquan Pei, Alex Podelko, Xudong Sun, Yi Wu, Erez Zadok, and Siyuan Zhou for invaluable discussions and feedback. Yoo is supported in part by the Siebel Scholar award. Xu's group is supported in part by NSF grants CCF-1816615, CCF-2029049, CNF-1956007, and a Facebook Distributed Systems Research award. The evaluation is supported by Microsoft Azure credits and Google Cloud credits.

REFERENCES

[1] Manage Chained Replication. https://docs.mongodb.com/manual/tutorial/manage-chained-replication/.

[2] Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. Cooperative Task Management without Manual Stack Management. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'02)* (June 2002).

[3] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)* (Oct. 2003).

[4] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. Fail-Stutter Fault Tolerance. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS'01)* (May 2001).

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

[6] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (Dec. 2004).

[7] Boucher, S., Kalia, A., Andersen, D. G., and Kaminsky, M. Lightweight Preemptible Functions. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'20)* (July 2020).

[8] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation (OSDI'06)* (Nov. 2006).

[9] Chandra, T. D., Griesemer, R., and Redstone, J. Paxos Made Live – An Engineering Perspective. In *Proceedings of the 26th annual ACM symposium on Principles of Distributed Computing (PODC'07)* (Aug. 2007).

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

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

[12] Cunningham, R., and Kohler, E. Making Events Less Slippery With eel. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS'05)* (June 2005).

[13] Curtsinger, C., and Berger, E. D. COZ: Finding Code that Counts with Causal Profiling. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[14] Dabek, F., Zeldovich, N., Kaashoek, F., Mazières, D., and Morris, R. Event-Driven Programming for Robust Software. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop* (July 2002).

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

[16] Gunawi, H. S., Suminto, R. O., Sears, R., Golliher, C., Sundararaman, S., Lin, X., Emami, T., Sheng, W., Bidokhti, N., McCaffrey, C., Srinivasan, D., Panda, B., Baptist, A., Grider, G., Fields, P. M., Harms, K., Ross, R. B., Jacobson, A., Ricci, R., Webb, K., Alvaro, P., Runesha, H. B., Hao, M., and Li, H. 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)* (Feb. 2018).

[17] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

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

[19] Huang, P., Guo, C., Lorch, J. R., Zhou, L., and Dang, Y. Capturing and Enhancing In Situ System Observability for Failure Detection. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (Oct. 2018).

[20] Huang, P., Guo, C., Zhou, L., Lorch, J. R., Dang, Y., Chintalapati, M., and Yao, R. Gray Failure: The Achilles' Heel of Cloud-Scale Systems.

In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)* (May 2017).

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

[22] Isard, M. Autopilot: Automatic Data Center Management. *SIGOPS Operating System Review 41*, 2 (Apr. 2007), 60–67.

[23] Jha, S., Cui, S., Banerjee, S., Xu, T., Enos, J., Showerman, M., Kalbarczyk, Z. T., and Iyer, R. K. 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)* (Nov. 2020).

[24] Krohn, M., Kohler, E., and Kaashoek, M. F. Events Can Make Sense. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'07)* (June 2007).

[25] Lamport, L. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32*, 4 (Dec. 2001), 51–58.

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

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

[28] Lou, C., Huang, P., and Smith, S. 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)* (Feb. 2020).

[29] Mace, J., Roelke, R., and Fonseca, R. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

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

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

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

[33] Ousterhout, J. Why threads are a bad idea (for most purposes). In *Presentation at the 1996 USENIX Annual Technical Conference* (Sept. 1995).

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

[35] Schultz, W., Avitabile, T., and Cabral, A. Tunable consistency in MongoDB. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB'19)* (Aug. 2019).

[36] Tan, C., Jin, Z., Guo, C., Zhang, T., Wu, H., Deng, K., Bi, D., and Xiang, D. 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)* (Feb. 2019).

[37] van Renesse, R., and Schneider, F. B. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (Dec. 2004).

[38] Wilcox, J. R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M. D., and Anderson, T. 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).

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

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

[41] Zhang, Q., Yu, G., Guo, C., Dang, Y., Swanson, N., Yang, X., Yao, R., , Chintalapati, M., Krishnamurthy, A., and Anderson, T. 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)* (Apr. 2018).