DepFast: Orchestrating Code of Quorum Systems

Xuhao Luo, *University of Illinois at Urbana-Champaign;* Weihai Shen and Shuai Mu, *Stony Brook University;* Tianyin Xu, *University of Illinois at Urbana-Champaign*

https://www.usenix.org/conference/atc22/presentation/luo

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

DepFast: Orchestrating Code of Quorum Systems

Xuhao Luo[†], Weihai Shen[‡], Shuai Mu[‡], Tianyin Xu[†]

[†]University of Illinois at Urbana-Champaign [‡]Stony Brook University

Abstract

Quorum systems (e.g., replicated state machines) are critical distributed systems. Building correct, high-performance quorum systems is known to be hard. A major reason is that the protocols in quorum systems lead to non-deterministic state changes and complex branching conditions based on different events (e.g., timeouts). Traditionally, these systems are built with an asynchronous coding style with event-driven callbacks, but often lead to "callback hell" that makes code hard to follow and maintain. Converting to synchronous coding styles (e.g., using coroutines) is challenging because of the complex branching conditions. In this paper, we present Dependably Fast (DepFast), an effective, expressive framework for developing quorum systems. DepFast provides a unique `QuorumEvent` abstraction to enable building quorum systems in a synchronous style. It also supports composition of multiple events, e.g., timeouts, different quorums. To evaluate DepFast, we use it to implement two quorum systems, Raft and Copilot. We show that complex quorum systems implemented by DepFast are easy to write and have high performance. Specifically, it takes 25%–35% fewer lines of code to implement Raft and Copilot using DepFast, and the DepFast-based implementations have comparable performance with the state-of-the-art systems.

1 Introduction

Quorum systems are critical distributed systems. In a quorum system, a node sends a request to a group of nodes, and proceeds on receiving a quorum of acknowledgements. The quorum size depends on the system design; it usually varies between a majority, a super-majority, or the whole group. The most common quorum systems are replicated state machines—a linearizable and fault-tolerant group of distributed nodes coordinating through a consensus protocol such as Paxos [39] and Raft [48]. Such quorum systems are widely deployed in practice, especially as critical infrastructures of large-scale cloud and Internet services [17, 20, 23, 34, 60].

Building a quorum system is hard. Quorum systems often have a complex consensus protocol. A node in these protocols have a complex state space. An event, e.g., a reply to a message, or a timeout, will trigger a state transition of the node. At each state, the node has multiple possible branches to go into based on the event and its current state. Though the state transition itself could be deterministic, the entire node behavior is not. This is mainly because of: 1) non-determinism in some event types, e.g., timeout, 2) the inter- and intra-node parallelism, and 3) network asynchrony, e.g., message delays and out-of-order delivery.

The traditional way to code these complex state transition conditions is through an event-driven or asynchronous coding style. For each event, the developer defines an event handler, or a callback function, to drive the program to the next state. On the one hand, writing code in this style could have its benefits, mainly two-folded: 1) it could match the style of a more formal description, e.g., a TLA+ [38] specification; 2) it could have a high performance as even-driven programming is often considered to be fast in concurrent programs.

On the other hand, coding in an asynchronous style has drawbacks, mainly making the code harder to write and follow, more error-prone, and harder to debug. Since the main workflow of handling a request from begin to end is expressed in many (callback) functions as opposed to a single function, the developer needs to manually maintain the shared control, data, and debug variables across these functions. The developer also needs to manually map each function execution properly to the request's lifetime, for example, dropping a reply if the system has moved with a quorum without the reply. Overall, this asynchronous code style could increase difficulty to develop and maintain the system. This problem is also known as stack ripping [15] or callback hell [28]).

A natural way of resolve the above issue is turning the asynchronous code into synchronous style, using lightweight solutions like coroutines. This is a well studied idea in systems research [15, 37, 52] and has been adopted widely in practice [13, 14]. However, the past study of using coroutines to inline callbacks mainly considers supporting a single asynchronous call [15, 52], e.g., inlining a single RPC callback. This is insufficient for a quorum system, which often has many concurrent callbacks and timeouts, and branches based on these concurrent event composition. Take the following behavior of a classic consensus system for example, a node sends requests to a group, then it needs to proceed to different branches based on the replies when it receives 1) a majority of acknowledgements, 2) a majority of rejects, 3) a majority of replies, mixed acknowledgements and rejects, and 4) fewer than a majority of replies. Each of these conditions have their own timeouts, and the program can only enter one branch. It is prohibitively difficult to express these conditions with a single coroutine. To make matters worse, the behavior could be even more complex with advanced protocols which have

multiple quorum sizes. For example, in fast-path enabled protocols such as Copilot [47], a super-majority quorum and a majority quorum apply at the same step in the protocol, which further complicates the situation.

In this paper, we present DepFast (Dependably Fast), a coroutine-based distributed programming framework to address the aforementioned challenges. Like prior works [15, 52], DepFast promotes synchronous code style with cooperative task scheduling and provides an Event abstraction to wrap the waiting points. The unique part of DepFast is that it provides a QuorumEvent abstraction that enables the construction of straightforward protocol descriptions, even for quorum systems with complicated timeout rules and multiple quorums (which was previously only expressible in bug-prone callback style). A QuroumEvent represents the system state of a quorum and any event that affects the state (e.g., arrival of a reply, a timeout) is funneled through the QuorumEvent. The program thus can synchronously express the conditions and branches using the QuorumEvent in a plain if-else style. Furthermore, the events are composable in DepFast, making it easier to deal with cases such as waiting for multiple quorums of different sizes at the same step.

Using DepFast to build quorum systems, the main control flow of the protocol can be written in a single function with an unripped stack. The code written in this style is easy to follow and to debug. As a case study, and as a motivation, we studied the fail-slow behavior of quorum systems, where a node in the system can be much slower than a non-faulty node but still functioning [55]. Debugging fail-slow behavior requires carefully identifying the line of code to add log statements for tracing the lifetime of a request. Writing code in the DepFast style simplifies debugging.

To evaluate DepFast, we use DepFast to implement two quorum systems based on Raft [48] and Copilot [47] respectively. We report our programming experience with DepFast. We demonstrate examples illustrating that DepFast leads to better implementations compared to the common practice of asynchronous, callback-style code; in particular, DepFast enables direct translation of the protocol algorithms and allows precise expressions of complex wait conditions, resulting in 25–35% fewer lines of code. We evaluate the performance of DepFast-based Raft and Copilot implementations against state-of-the-art versions to show our approach imposes no performance penalty. Moreover, the DepFast-based implementations have better tolerance against various types of fail-slow faults by construction, than the state-of-the-art versions.

The paper makes the following contributions:

- We design QuorumEvent to enable the construction of protocol descriptions even for quorum systems with complicated timeout rules and multiple quorum sizes.
- We develop DepFast, an effective, expressive framework that enables developers to implement complex quorum systems with a synchronous programming style.
- We show how DepFast benefits the implementation of quorum systems by illustrating DepFast-based Raft and Copilot implementations compared to state-of-the-art versions.
- We evaluate the performance of DepFast-based Raft and Copilot and show that the DepFast design for the ease of programming does not come with a performance penalty.

2 Background and Motivation

2.1 Quorum systems and async. programming

This paper uses quorum systems to refer to distributed systems with a communication pattern that requires replies from a quorum in a group of nodes. One typical type of quorum systems is a replicated state machine which uses consensus protocols (e.g., Paxos [39] and Raft [48]) to let nodes agree on the next state transition.

Consensus protocols are known to be hard to implement. Take the following behavior in Paxos and Raft for example, a node broadcasts a replication request (Accept in Paxos, or AppendEntries in Raft) to a group; then the node needs to react to events including replies and timeout. A reply can be either an acknowledgement or a reject. Based on different events, this node needs to enter different branches when it receives: 1) a majority of acknowledgements before the timeout, 2) a majority of rejects before the timeout, 3) a majority of replies, mixed acknowledgements and rejects at the timeout, and 4) fewer than a majority of replies at the timeout. After the node chooses a branch, any future event needs to be dropped. The inter- and intra-node concurrency and the non-deterministic event triggering lead to a very complex program state space.

Many formal and informal protocol descriptions of quorum systems are written in an *asynchronous* style: "upon receiving a (reply to a) message, the system acts as follows." It is intuitive to construct the code in the same way (e.g., writing a message handler in the message loop, a callback in event-driven model, or an actor in the actor model to process a particular message). Coding in the asynchronous style has the benefit of matching a formal description of the protocol (e.g., a TLA+ [38] specification). However, it could cause the control flow to be shredded into many sub-functions like callbacks. This leads to spaghetti code, also known as stack ripping [15] or callback hell [28]. Take a Paxos system for example. For each request that goes through the 3 phases (Prepare/Accept/Commit), the main control flow will at least be shredded into 3 types of callbacks. If this is a 5-replica system, the callbacks will be executed 3×5 times. If we further count callbacks caused by disk logging (asynchronous I/O), there will be even more (at least doubled) callbacks.

It does not take long before a developer loses track of how callbacks affect each other. It also imposes significant challenges to manage the waiting process, especially in cases where the callbacks are written by different developers (which is often the case in practice). A natural way to address this

problem is to turn the code to a synchronous style by inlining the callback into the calling function. This could greatly improve understandability and maintainability. In fact, some papers describe the main control flow in this style. For example, in the Paxos paper [39], after a proposer sends out the proposal, "*if the proposer receives a response from a majority of acceptors...*" In practice, many projects started with the asynchronous callback style, but the growing size of codebase and the accompanying cognitive load of callbacks made the developers change to a synchronous code style [15, 18].

In the past, asynchronous versus synchronous programming styles (or event versus thread) have drawn many discussions [15, 19, 25, 26, 37, 44, 49]. The asynchronous or event-driven style is often preferred for performance reasons. For example, it can avoid the overhead of many OS threads, and the `epoll/kqueue` model of processing network interrupts can efficiently utilize the interrupt-based OSes and devices. Past works have proposed solutions to turn asynchronous code into synchronous and preserve its high performance, e.g., by using lightweight threads like stackful coroutines [15, 52]. However, existing solutions mainly consider inlining a single callback like a single RPC. It does not address the challenges of implementing quorum systems, where a broadcast request triggers a group of callbacks, and the system needs to proceed with a quorum.

2.2 Experience in debugging fail-slow behavior

We recently studied fail-slow behavior of real-world quorum systems [55]. Our journey started from observing that quorum systems of product-grade databases cannot meet fault-tolerance properties of the consensus protocols—a fail-slow follower affects system-wide performance. Such behavior contradicts the theory—a quorum system should proceed when there is a majority of non-faulty nodes. Specifically, a fail-slow follower should not have visible impact by design.

To reveal the root causes of the fail-slow behavior, we spent two person-years to analyze the three implementations. In our experience, debugging fail-slow fault tolerance is challenging and time-consuming. At a high level, the debugging process is a binary search for small fragments of code that caused the slowness using time stamping. The process sounds easy (as we imagined it to be), but is painful in practice. The asynchronous code often looks like spaghetti: the main control of a request is spread in different code fragments. Understanding where those code fragments are located and how they interact is non-trivial. In fact, working with developers of two of the databases, we find that even they have the same experience.

We also find that asynchronous code often lacks a clear abstraction between the quorum logic (e.g., the Raft protocol) and common, low-level utilities (e.g., RPC, disk I/O) in the spoken implementations. In addition to the challenge of expressing complex state transitions, lacking a clear abstraction has two more problems in implementation.

First, when a buggy fail-slow behavior occurs, it is hard to know whether the bug is caused by the protocol code or the utility code. A bug in the utility code is typically easier to identify and fix than a logic bug. It would be very helpful if the code can be constructed in a way that the logic code and utility code are isolated and separately profiled for debugging.

Second, a lack of abstractions also indicates lacking knowledge across the two parts. The utility code has to blindly execute the requests passed by the logic code and cannot perform optimizations to tolerate fail-slow behavior, but push the burden back to protocol logic. For example, the Raft logic broadcasts AppendEntries to all replicas and waits for a quorum of replies to proceed. In many existing implementations, the Raft logic sends the same message to each replica and the utility faithfully puts the message to the buffer of each replica. If one replica is slow, the connection would be slow and the buffer would keep increasing, leading to the backlog issue reported by recent work [27, 29]. If the utility 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.

2.3 Goal

Our experience in building and investigating complex quorum systems has driven us to rethink the programming practice and seek a more foundational solution that can make it easier to build and maintain quorum systems. We propose a synchronous programming framework for this purpose. In particular, we will propose abstractions that allow developers to implement complex quorum state transitions that can only be implemented with an asynchronous programming style before. We will show by our experiences that this framework can help programmers efficiently express complex quorum conditions in a way that is easy to follow and maintain.

3 The DepFast Framework

This section introduces the Dependably Fast (DepFast) framework. DepFast aims to provide an effective, expressive programming framework for building quorum systems. We first go through the interface of DepFast (§3.1), including an important abstraction we propose for quorum systems (§3.1.2). Then we go into internals of the framework, describing how it is implemented (§3.2).

3.1 DepFast from a programmer's perspective

3.1.1 Coroutines and events

DepFast provides programmers with two main interfaces:

- a coroutine interface for launching tasks,
- an event interface which wraps the waiting points, or any potential fail-slow points, in the code.

The idea of using coroutine is to keep the code in one piece—avoiding callbacks, while maintaining the performance when dealing with an operation that needs to wait. For example, the code snippet below is triggering an RPC with a callback. Traditionally, it is considered the high-performance way of writing concurrent programs as it avoids the cost of creating and switching between threads. On the other hand, it comes with the cost of breaking the control flow, leading to many side effects besides increasing code complexity. For one, the programmer needs to manually maintain a shared stack from callbacks to callbacks, which is also known as *stack ripping* [15]. For a quorum-based system, the case could be more complex. For example, a reply is not always "valid"; an outdated reply needs to be ignored. The programmer needs to manually manage those, as exemplified in the code below:

```
void DoAppendEntries() {
  for (auto rpc_proxy : servers) {
    auto entries = ...;
    // the next line bears possible slowness
    auto rpc_event = rpc_proxy.AppendEntries(entries,
        AppendEntriesCallback);
  }
}
void AppendEntriesCallback(Id id, Result result) {
  // manually manage shared data
  auto reply_ok_cnt = reply_map_g[id];
  // manually manage lifetime
  auto status_ = status_map_g[id];
  if (status_ == undecided) {
    ... // only process when the log is still alive
  } // else ignore
}
```

Using coroutine, the above code can be expressed as:

```
Coroutine::Create([] () {
  vector<RpcEvent> events;
  for (auto rpc_proxy : servers) {
    auto entries = ...;
    // the rpc call is asynchronous and non-blocking
    auto rpc_event = rpc_proxy.AppendEntries(entries);
  }
  for (auto& rpc_event: events) {
    // block coroutine until the rpc returns
    rpc_event.Wait(); // possible slowness
    Process(rpc_event.Result());
  }
})
```

In the above code, what was split in callbacks is glued together in a single function, where the programmer can continue using the stack to share system states with a unified control flow. The code using coroutine can provide similar performance to the code with callbacks, because coroutines do not incur the heavy overhead of OS threads [15].

Note that the above code uses `RpcEvent`: calling `Wait()` on this object would suspend the current coroutine until the RPC has the return value ready. DepFast uses such event abstraction to wrap all the waiting conditions. With the events, a programmer can suspend/resume the coroutines. DepFast implements many built-in event types to support various operations (RPC, file I/O, etc.). With all waiting points moderated by the framework, DepFast naturally empowers more analysis (§A.2). DepFast also provides novel event types to better deal with distributed, quorum-based systems. We next introduce an important event in DepFast, `QuorumEvent`.

3.1.2 QuorumEvent

The coroutine-style code above is less efficient than the callback-style code. The coroutine-style code waits for the reply from each server sequentially with a fixed order, while in the callback-style code the callback for each server's response can be triggered out of order. This gives the callback-style code a major benefit in performance, because it only needs to wait for the first quorum of messages to arrive. In particular, it is not affected by a fail-slow remote server. On the contrary, the coroutine-style code will be slower in performance and be affected by a fail-slow server.

To address this issue, DepFast has a special event type, termed `QuorumEvent`. The key idea is to prevent any individual fail-slow event from straggling a coroutine by combining many events together into a compound event. As the name suggests, `QuorumEvent` does not need all responses from each individual event. The usage of a `QuorumEvent` is demonstrated by the following example.

```
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
  }
  quorum_event.Wait(MAJORITY); // wait for a majority
  ...
  // after 10s, release resources to avoid backlog
  quorum_event.Release(10000 /*ms*/);
})
```

Using the above `QuorumEvent` has two benefits. First, the replies can be received out of order and the program can proceed as soon as receiving a majority of replies. Second, the abstraction helps programmers avoid writing any code that would be blocked by any single-point fail-slow remote server. A key idea that DepFast deploys to help avoid fail-slow fault propagation is to encourage programmers to not wait on any event individually but always wait on a quorum of events when possible.

`QuorumEvent` also provides a better abstraction for resource management to help avoid the backlog issue constantly observed with fail-slow faults [27, 29]. In the code snippet above, calling `Release()` will tell DepFast to release all resources related to this event after a timeout. This is particular useful for a quorum system. It gives a deferment period for the slower response to arrive, after which the system will forcibly free all the resources related, e.g., the buffer in the RPC. Allowing a deferment period is very useful in implementing many protocols, because they often need to distinguish between the two cases: 1) the response is only a bit slow, or not at all slow but just ordered after other responses, but the host can still react to those responses which benefits the system liveness 2) the response is too slow, and the receiver should react differently (e.g., a failure recovery).

To see why using `QuorumEvent` is a better approach for resource management, consider the following alternative:

```
// Create a separate coroutine for each RPC
for (auto rpc_proxy : servers) {
  Coroutine::Create([]() {
    auto entries = ...;
    auto rpc_event = rpc_proxy.AppendEntries(entries);
    rpc_event.Wait();
    Process(rpc_event.Result());
  })
}
```

This alternative approach is both synchronous in programming style and avoids sequentially waiting on each RPC call. Actually, such a-coroutine-per-task approach is very popular in modern practices, especially in writing services with Go-lang [22]. However, in our experience this may cause problems under fail-slow behavior. If the target of the RPC is slow and cannot respond in time, the coroutine will hang in the system, waiting for the response. With new requests coming in, the hanging coroutines will accumulate, eventually exhausting system resources, as each coroutine consumes at least a memory space for its stack.

3.1.3 Other event types

In general, DepFast provides two types of events: basic events and compound events. Basic events are waiting on a task to finish, such as an RPC, a disk access, or a flag to be set, etc. A compound event is a combination of basic or other compound events. Table 1 summarizes common event types in DepFast and their trigger conditions.

Basic events. One common basic event is `ValueEvent`. This is a holder for a value to be set. If the value is set to match the target value, the event will be triggered. We find this event abstraction very useful because we often find statements in algorithms written like "*wait for X to become Y.*" For example, in Copilot [47], to decide the execution order of a command, the system needs to wait until the status of a selected group of commands to become "committed". Traditionally, this could be hard to implement as it would involve complex callbacks or thread synchronization. With DepFast, statements of this type can be directly translated to one line of code.

Another basic event type is `IOEvent`. We use it to wrap all synchronous I/O operations, mainly disk-related operations. An `IOEvent` corresponds to a task executed in an I/O thread. For example, the program initiates a disk write through DepFast's interface, the actual disk operations involving `fwrite` and `fsync` will be executed in the I/O thread. The program then waits on a `DiskEvent` returned by DepFast. When the disk operation finishes, the I/O thread will notify the scheduler that the event is ready. The synchronization between I/O threads and scheduler is the only part in a DepFast program that has multi-thread synchronization. We believe that this small footprint of multi-thread synchronization can minimize the possible fail-slow issues caused by multi-threading.

Compound events. For compound events, an example is `QuorumEvent` (§3.1.2). It takes many events (e.g., `RPCEvent`) as its subevents, and wait for at least a defined quorum of them to be triggered. Other compound events in DepFast includes `AndEvent` and `OrEvent`. As the name suggests, an `AndEvent` is triggered when all of its subevents are triggered; an `OrEvent` is triggered as soon as one of its subevents is triggered. Note that events can be nested: for instance, an `AndEvent` can contain many `QuorumEvent`s as its subevents.

Event	Trigger Condition
`ValueEvent`	If the value is set and matches the target (it supports customized comparators).
`DiskEvent`	If the disk access operation (e.g., `fread`, `fwrite`, and `fsync`) is finished.
`RpcEvent`	If the RPC call has returned.
`QuorumEvent`	If a quorum has reached (typically used together with `RpcEvent`).
`AndEvent`	If all subevents have been triggered.
`OrEvent`	If any subevent is triggered.

Table 1: The built-in events in DepFast

Nesting events can express complex waiting conditions. For example, one can use an `OrEvent` to combine these 3 events: 1) a `QuorumEvent` that waits for a majority of okays. 2) A `QuorumEvent` that waits for a minority-plus-one rejects. 3) A `TimeoutEvent`. This compound event can be used to effectively catch conditions in classic consensus protocols. In fact, as we find that the abstraction is very commonly used, we have merged these three conditions into `QuorumEvent` so it has three outcomes: `Ready(), Fail(), Timeout()`,

3.1.4 A showcase of DepFast's expressiveness

DepFast can effectively express many complex behaviors of quorum systems, organize the main control flow in a clean way, and process the complex state transitions automatically in the background. We demonstrate the expressiveness of DepFast using the code snippet of our Copilot implementation built with DepFast in Figure 1. Copilot is one of the protocols that leverages a "fast-path quorum" [40, 45, 57]. In these protocols, after broadcasting a round of (FastAccept) messages, there are at least three concurrent conditions to decide how the system proceeds: 1) to a fast path if receiving a super-majority of acknowledgments with identical speculative information, 2) to a slow path if receiving a majority of acknowledgments, and 3) to failure recovery if neither the above is possible (e.g., when receiving a majority of rejects, or not enough messages after a timeout). Figures 1(a) and (b) show the code and control flow chart of Copilot built on top of DepFast. As demonstrated, the code implements the protocol in a clean manner. What DepFast handles in the background is shown in Figure 1(c). Upon every event (message arrival and timeout), DepFast processes the subtle state transitions and drives the main control flow forward to the proper next state. Without DepFast, one needs to carefully implement all state transitions and error handling manually, a complex and error-prone process. With DepFast, one can build the system cleanly, with code easy to follow.

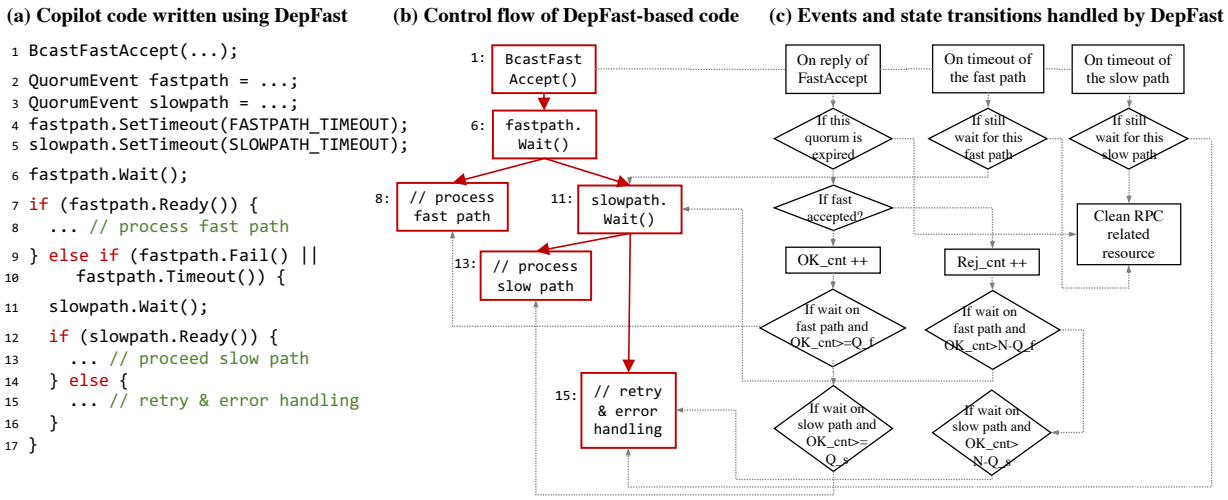


Figure 1: Expressiveness of DepFast demonstrated by DepFast-based Copilot implementation: (a) code, (b) control flow, and (c) events and state transitions. The code logic is explained in Figure 3.

3.2 DepFast internals

3.2.1 Architecture

Figure 2 shows the architecture of DepFast. A DepFast process has two major types of threads: worker threads and system threads. The former run user code and the latter run background activities.

All user-defined tasks run in worker threads as stackful coroutines; we use Boost::coroutine2 [13] as a building block. A worker thread runs an `epoll` [6] (or `kqueue` [10] for BSD-based systems) loop that wakes up on network interrupts or timeouts. The system has a built-in high-performance RPC module that uses the `epoll` for incoming and outgoing messages similar to other high-performance networking library like libev [11]. The RPC module provides automatic client code and server handler header generation from an RPC declaration file (like gRPC [9]). The RPC works asynchronously and provides synchronous event binding (RpcEvent).

Having the RPC working asynchronously allows us to avoid launching a separate coroutine for each RPC (§3.1.2). Take the `QuorumEvent` for example. The system only has one global `epoll` loop that receives replies for all RPCs. Once the system receives a reply, it is matched to the belonged `QuorumEvent`, and a counter is updated in that `QuorumEvent`. Once that counter reaches the quorum number, the `QuorumEvent` is ready, and then the system resumes the coroutine waiting on it. There is no coroutine creation during handing a message that might contribute to a quorum.

In the same worker thread runs the scheduler functions. The scheduler is in charge of managing user coroutines. All coroutine lifetime related functions, including coroutine creation, deletion, pause, and resume, are provided by the scheduler. The scheduler performs a check at each `epoll` wakeup. It checks whether the event a coroutine is waiting on is triggered

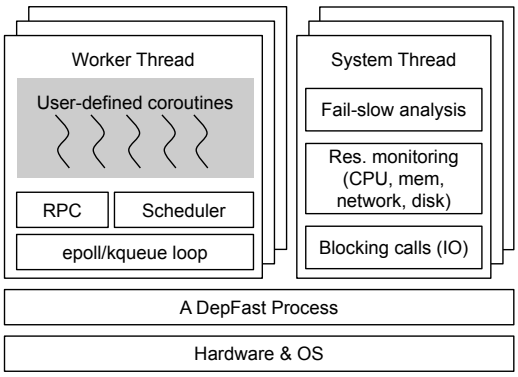


Figure 2: The architecture of DepFast

(or timed out), and then resume the paused coroutine.

While the worker threads handle most of the system functions on the critical path, the system threads deal with the additional features of the framework. Among these, a significant function is to support blocking calls. Examples include disk flush calls (`fsync`) and support for third-party libraries. DepFast supports these by wrapping them in system threads. Other features that run in system threads include the resources usage monitoring (CPU, memory, network, disk) and fail-slow analysis, which we will discuss later.

3.2.2 Lazy and cooperative scheduling

The base class of an `Event` is implemented as follows:

```
class Event {
  // timeout defines the maximum duration of waiting
  // on the event. -1 means waiting forever.
  void Wait(int timeout = -1) {
    if (IsReady()) return;
    if (timeout >= 0) {
      sched_s.sorted_timeouts.insert(now()+timeout);
    }
    // sched_s is a thread_local static variable
    sched_s.yield_current_coroutine(this);
  }
```

```
  void Test() {
    if (IsReady()) {
      // the scheduler will put the suspended coroutine
      // back to the ready_coroutine queue.
      sched_s.notify_ready(this);
    }
  }
  // To implement by specific event type to define when
  // this event is ready to resume suspended coroutine.
  virtual bool IsReady();
}
```

The coroutine management operates in a lazy and cooperative approach. It does not preemptively suspend or resume a coroutine. When a coroutine is running, it fully occupies the worker thread. A key structure in the scheduler is a queue that records pending events and the corresponding suspended coroutines. When a coroutine runs, it may change the status of other events (e.g., changing the value of `ValueEvent`). When this happens, the scheduler will *not* yield the current running coroutine and switch to the resumable coroutine immediately, but mark the event ready and put it in a ready queue. Only when the running coroutine calls `Wait` on an event, the scheduler is triggered to do scheduling work.

The scheduler will first check if the event is ready (i.e., no wait is necessary). If so, the `Wait` call returns directly without yielding out the coroutine; the caller coroutine continues running. If not, the scheduler will put a reference of the event and the caller into a pending queue, and then looks for the next coroutine to run. The scheduler first checks the queue of ready events and resumes them one at a time. Note that the queue of ready events may grow during this process, as the resumed coroutine may turn more events ready. After the ready queue is empty, the scheduler searches the pending queue for events that are timed out, and resumes the suspended coroutines correspondingly. To make the search faster, the pending queue is sorted based on the timeout timestamps.

Reversed backlog. In our development of DepFast, we encountered an interesting issue termed "reversed backlog." When a system has a slow node, RPCs to this slow node will be delayed. However, the delays are not necessarily uniform over time. The responses often arrive in a burst pattern—the RPC sender will not hear back from the slow node for a while, and then suddenly receive many responses from it. In our early implementation, the epoll loop would process everything available from a connection before moving on to the next. This caused the system to occasionally hang on processing the outdated RPC responses. To deal with this issue, the system is improved to process data from connections with a round-robin approach; it will pause processing a connection after reaching a threshold to avoid starving other connections.

3.2.3 Concurrency and multi-threading

Through the coroutine model, DepFast allows multi-task concurrency inside a single worker thread. Similar to many other asynchronous event or coroutine frameworks, DepFast encourages programmers to exploit concurrency in a single worker thread before moving to multi-threading. The advantage of running tasks in a single thread is to avoid thread-safety issues. To DepFast, an extra benefit is to eliminate the possibility of fail-slow faults caused by thread locks, a known suspect of performance issues. In reality, running tasks concurrently with a single thread can give high enough performance in most cases, as shown in our evaluation (see §6.2 and §6.3).

Moving to multiple threads, to write thread-safe code the users need to either shard the system into different threads and regulate inter-thread communication, or write memory-sharing code and use mutexes for mutual exclusion. DepFast encourages the former as it minimizes the chances of performance issues caused by waiting on mutexes.

4 Discussion

We discuss the software engineering benefits of using DepFast to build quorum systems. We exemplify the challenges in implementing complex quorum conditions and discuss how DepFast can address the challenges. As an concrete example, the following code snippet processing a fast path is taken (and simplified) from the EPaxos implementation [3], a popular academic prototype of advanced consensus protocols.

```
func handlePreAcceptReply(reply) {
  inst := ... // find consensus instance
  ... // return if this is a delayed request
  if reply.OK {
    inst.preAcceptOKs++
  } else {
    inst.preAcceptRejects++
    if inst.preAcceptRejects >= r.N/2 {
      // TODO
    }
  }
  fastpathSatisfied = ... // test fastpath conditions
  if inst.preAcceptOKs >= N/2 && fastpathSatisfied {
    // proceed to fast path
  } else if inst.preAcceptOKs >= N/2 {
    // proceed to slow path
  }
  //TODO: take the slow path if msgs are slow to arrive
}
```

We can see two `TODO`s in the code. The first `TODO` is on counting rejects. If a node receives too many rejects to proceed, it should enter error handling to retry this request. In DepFast, this maps to the branch where `slowpath.Fail()` happens. The second `TODO` is when messages are slow to arrive, this case maps to DepFast's branch where `fastpath.Timeout()` and/or `slowpath.Timeout()` happen.

Implementing the two `TODO`s would require heavy revisions on the code logic and change the control flow, as it cannot be done by naturally replacing the `TODO` comments with two function calls. In contrast, with DepFast, one can implement both `TODO`s in place. For the first `TODO`, a retry function can be synchronously called in the reject branch. For the second `TODO`, if messages are slow to arrive, a function calling the slow path can be put right in the timeout branch.

In fact, `TODO`s are not the only problem in the code. The fast-path condition is also simplified and suboptimal. The

condition is important, because it decides whether to enter a fast path or a slow path. In the code, once it sees a majority of OK replies, it makes the decision based on the current calculation of the conditions (`fastpathSatisifed`). However, this decision could be suboptimal, because the conditions could change if more replies arrive. As a consequence, a node loses the opportunity to enter a fast path if it waits a little longer, but directly enters the slow path. The simplification, despite being suboptimal, is understandable, because it is very hard to express the conditions accurately in the asynchronous code style. With DepFast, this can be expressed easily with the timeout scheme on a QuorumEvent.

Lastly, the conditions assume that the fast- and slow-path quorums have equal size, which indicates that it only works for at most 5 replicas. If the replication group size is bigger [4], the code needs heavy revisions. Simply replacing `N/2` in the `if` branch with a larger super-majority value is incorrect: the code will always choose a slow path because the `else if` branch will always be taken.

Although this discussion is based on EPaxos, we find that the conditions are error-prone in other quorum system implementations, especially regarding timeout handling. For example, CockroachDB had a bug caused by having no timeout on lease acquisition [5], which may lead to system stalling. In DepFast, because timeout can be easily added to an event, such problems can be prevented. In fact, we often use this practice for debugging: for cases that cause the system to stall, we add a global default timeout to all events, and then we can easily find the stall point of the problematic code.

5 Building Quorum Systems with DepFast

To demonstrate the usefulness and effectiveness of DepFast, we use DepFast to build two quorum-based systems: Raft [48] and Copilot [47], named as DepFast-Raft and DepFast-Copilot respectively. In this section, we discuss our experiences in using DepFast to build these systems, with a focus on how to "translate" the protocol algorithms into system implementations effectively.

5.1 DepFast-Raft

Raft's protocol largely consists of two parts: 1) *leader election*: a candidate broadcasts `RequestVote` RPCs to all the other servers; it becomes a leader once votes from a majority of servers are received. and 2) *data replication:* for each follower, the leader keeps a mark (`nextIndex`) of the next log position to send to that follower; if the follower is lagging (due to out-of-order messages, network issues, etc.), the leader repeatedly sends the log entries needed by the follower.

Leader election can be effectively expressed with DepFast's `QuorumEvent` design: a server broadcasts requests to other servers and can proceed after it receives a quorum of acknowledgments. Data replication, though described in a different style in the Raft paper [48]— from a follower's view, not a quorum's view—can also be expressed to the same pattern above. Our implementation uses one coroutine to initiate the broadcast of the `AppendEntries` requests and wait for a quorum of responses. As DepFast handles most of the complexity in the network, disk, and event processing, a Master student was able to translate Raft's pseudocode directly into a stable C++ implementation (used in §6.2) in ten days. The implementation has ∼1,200 lines of code. As a rough comparison, the Raft logic in etcd [8] is implemented in ∼1,600 lines of code in Go; braft [2], an open-source Raft implementation in C++, has ∼3,500 lines of code.

An interesting case we found in implementing DepFast-Raft relates to the way Raft describes its protocol. In the design of Raft's data replication protocol (its `AppendEntries` RPC), if the follower is lagging, the leader repeatedly sends log entries that are missing on the follower. This design is optimized for lagging servers and for new servers trying to catch. However, the way the algorithm is described may naturally lead to an implementation with separate threads synchronizing with different followers. A natural implementation could split the code responsible for committing a request into different functions in different threads. This style of implementation works well when there are no failures, but could take more time to debug when there are unexpected fail-slow behaviors, because it requires more work to trace the progress of each request. Our implementation, instead, uses a single coroutine to initiate the broadcast of the `AppendEntries` requests and wait for a quorum of responses. In case of an occasional reject due to the follower being lagging, the leader will launch a background coroutine, which is off the critical path of client requests, to synchronize with the lagging follower with additional `AppendEntries`.

5.2 DepFast-Copilot

Copilot is a consensus protocol that tolerates any single fail-slow node including the leader. It has two leaders, a pilot and a copilot, each is the backup of the other in case one fails. Copilot's complexities mainly rise from following designs:

- *Commands ordering.* Each leader maintains two separate logs, one for itself and one as the backup of the other leader. Copilot's ordering protocol coordinates between pilot and copilot to determine the *dependencies* of log entries, which specifies the prefix of the other log that should be executed before a given log entry.
- *Commands execution.* Unlike Raft that uses an index to order command execution, Copilot's execution order is more complex: (1) Copilot has a protocol that calculates the right order of a command based on its dependencies. The calculation process is restricted by the status of these dependencies, e.g., the execution must happen after all the dependencies satisfy a rule; (2) The execution of the command is also constrained by the dependencies, i.e., a command's execution

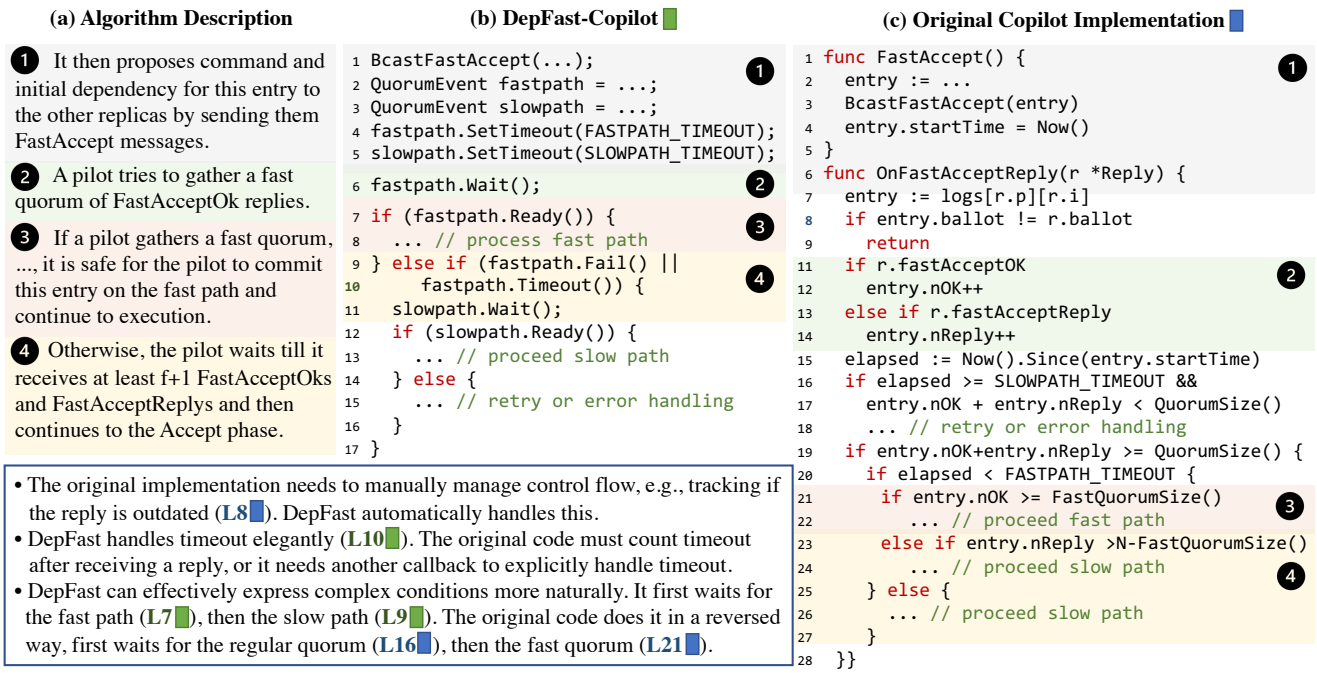


Figure 3: Comparison of the Copilot implementation using DepFast and the original implementation (both are simplified)

must wait until all the predecessors are executed.

- *Fast takeover.* When one pilot becomes slow or fails, the other pilot needs to take over entries in the slow pilot's log to prevent waiting for its commit for too long. It broadcasts `Prepare` to all other replicas to collect the entries and their status on other replicas at that position. Depending on the replies collected, Copilot distinguishes between many different cases to choose an entry properly, including whether there are committed entries, how many entries are fast-accepted, how many entries are accepted, etc.

As we have discussed in Section 3.1.4, DepFast can effectively express Copilot's complex behaviors. Figure 3 shows a comparison of the DepFast version and the original version of Copilot. As shown, the DepFast version is closer to the algorithm in flow, and is easier to follow.

The commands execution algorithm contains statements like "*waiting for the commit/execution of*", which is common in other protocols that use dependency for commands ordering (e.g., EPaxos [45]). However, such kind of behavior is not straightforward to express in an asynchronous programming style. In fact, the original Copilot implementation adopts this asynchronous programming style. It uses a separate Goroutine to keep scanning through the log and breaks the loop to start from the beginning when the above waiting condition is not satisfied. Instead, DepFast's `Wait` API captures such behavior effectively. What we do is to represent the commit/execution of a log entry as an event and call `Wait` if there is a dependency on it as shown in Figure 4.

We find that the Copilot implementation using DepFast is more concise and readable than the original asynchronous, callback-style implementation. To give a rough, unsolicited idea, the original implementation of the core protocol (excluding the utility code) has ∼2,500 lines of Go code [12]. DepFast-Copilot only has ∼1,600 lines of C++ code, despite that C++ is less expressive than Go.

Anecdotally, we started a Copilot implementation without DepFast, using an asynchronous callback style. In the process we ran into a bug that *sometimes* froze the system, which was caused by a wait condition being not triggered properly. We find bugs of this type are very hard to debug (we spent two weeks debugging it) because there is not a simple way to track each wait condition. We re-implemented the wait conditions using DepFast's `Wait` API. The new implementation was done in roughly two days and we never encountered the same problem. Thanks to DepFast's coroutine and event model, it is very easy to find out which event the system is waiting on, and print all the stack frames of the suspended coroutine.

6 Evaluation

We have shown the software engineering benefit of DepFast is its expressiveness and programmability (§3.1.4, §4 and §5). In evaluation, we mainly focus on answering two questions: 1) Does the expressiveness come at a cost of performance, or, can systems implemented in DepFast achieve the same level of performance as heavily optimized production and academic systems? 2) Can DepFast help system implementations guarantee their fault tolerance? This section answers the two questions by comparing Raft and Copilot implemented in DepFast to etcd and the original Copilot implementation. Specifically,

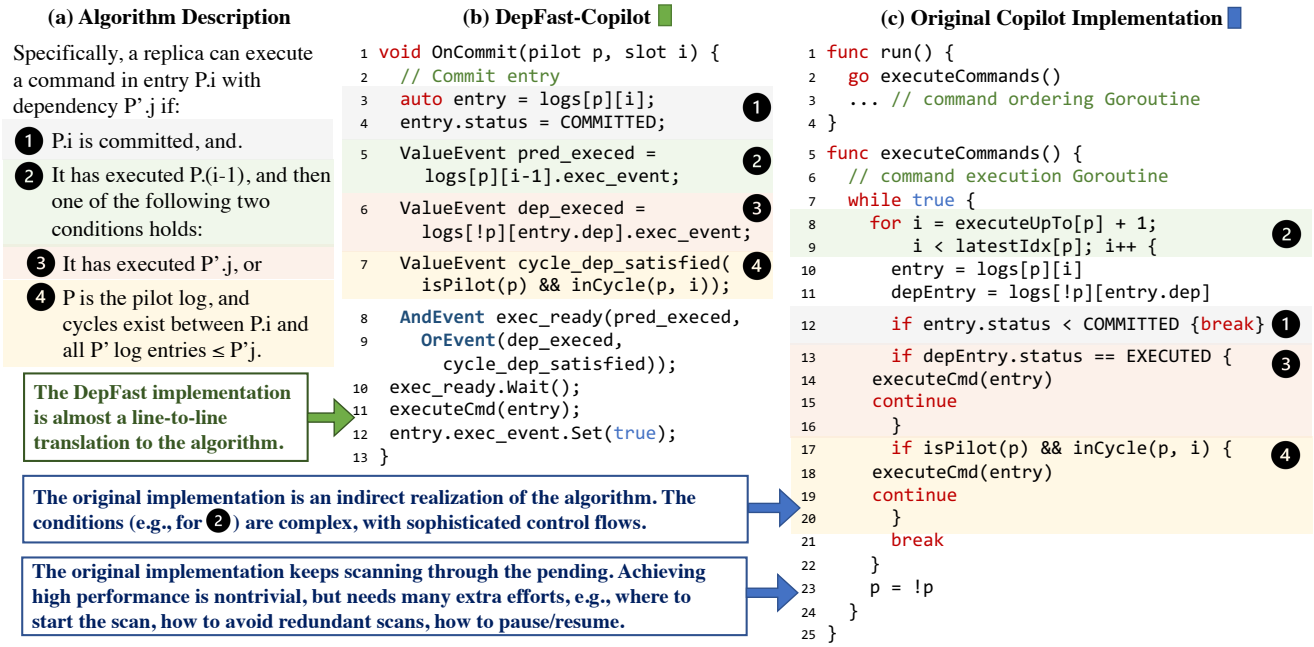


Figure 4: Comparison of the Copilot implementations using DepFast and the original impl. (both are simplified), cont'd

for fault tolerance, we evaluate DepFast-based systems on *fail-slow* fault tolerance using fail-slow faults [27, 31].

6.1 Experiment Methodology

We ran all the experiments on Azure Cloud. For each system, we evaluated it on a 3- and 5-replica cluster: each node runs on a Standard_D4s_v3 virtual machine (VM), with 4 vCPUs, 16GB RAM, and 64GB SSD. The server process is bind to one CPU core for all the systems evaluated. We ran the clients in a Standard_D16s_v3 VM, with 16 vCPUs and 64GB RAM.

Workloads and metrics. For all the evaluated systems, including our Raft and Copilot implementations, and the reference implementations (etcd [8] for Raft and the original Copilot implementation [12]), we use a single K-V, 100% write workload. We measure performance metrics, including throughput and latency distribution. Each trial runs for 120s and is repeated for 3 times. We display the results with the median throughput of the 3 trials, with the error bars showing the deviation.

Configurations. We use quorum reads and writes in our evaluation. We use the load that reaches the max CPU utilization on leader for the fault-injection experiments, marked with red stars in Figures 5(a) and 6(a). For DepFast-Raft, the server replies to the client after the log entry has been persisted on disk. For DepFast-Copilot, we set the fast-takeover timeout to 10ms and the command batching timeout to 1ms (same as the original Copilot implementation).

Fault Injection. We build a fail-slow fault injection testing tool to inject different types of fail-slow faults on system components (including CPU, memory, SSD, and network interface) into the target systems and measure their impact in terms of the end-to-end performance. The fail-slow faults are simulated based on prior studies on fail-slow faults and represent common fail-slow modes [27, 31]. Table 2 describes those faults and the corresponding injection methods. For DepFast-Copilot, we do not inject faults on the disk, because the implementation is memory-based.

We inject fail-slow faults in the way that is expected to be tolerated without losing throughput by the consensus protocols of the target quorum systems. For DepFast-Raft, we inject faults to a minority of followers [55]. For DepFast-Copilot, we inject faults to a minority of nodes that can include one of the leaders [47].

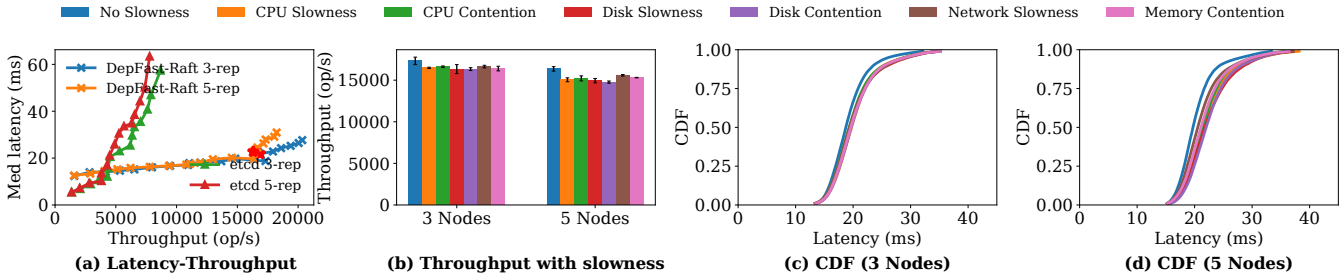
6.2 DepFast-Raft

Figure 5(a) shows the latency and throughput of DepFast-Raft and etcd with both 3- and 5-replica setups. When binding the server process to one core, our DepFast-Raft achieves a maximum throughput of over 20K and 18K RPS (requests per second) with 3- and 5-replica setups, respectively; etcd has a peak throughput of 8K RPS. Claiming DepFast-Raft is better than etcd would be perhaps unfair as etcd is a production system with many features. However, this test at least proves that DepFast can be used to implement systems of production-level performance, with a class-project level of building difficulty.

Figures 5(b)–(d) show the fail-slow fault tolerance of DepFast-Raft in terms of throughput, median and P99 tail latency (in CDF) in both 3- and 5-replica setups. For the 3-replica setup, we inject fail-slow faults to one follower; for the 5-replica setup, we inject fail-slow faults to two followers (the

Fail-slow Type	Injection Method	Follower	Leader
Slow CPU	Use `cgroup` to limit DB process to utilize only p of CPU period	p=5%	p=50%
CPU Contention	Custom program (to consume cpu) assigned $t\times$ cpu share as the DB process	t=15	t=1
Slow Disk	Use cgroup to limit the disk I/O bandwidth available for DB to bw	bw=128KB/s	N/A
Disk Contention	Use program(`dd`) to do write operation on disk while DB is running	no parameter	N/A
Slow Network	Add a delay of d to the network interface using `tc`	d=40ms	d=40ms
Memory contention	Use `cgroup` to set the maximum amount of user memory for DB process to s	s=50MB	s=250MB

Table 2: Fail-slow faults used in the measurement study and our evaluation



(a) Latency-Throughput **(b) Throughput with slowness** **(c) CDF (3 Nodes)** **(d) CDF (5 Nodes)**

Figure 5: Performance of DepFast-Raft with various fail-slow faults on follower(s) in the 3- and 5-replica deployments.

minority). We choose to only inject faults on the followers, because slow followers in Raft should not affect performance, while the slow leaders do.

We can see that DepFast-Raft consistently tolerates the injected fail-slow faults. The throughput differences are within 6% and 10% for 3- and 5-replica setup, respectively. The differences in both median and P99 latency are within 15% range. The results are comparable to etcd under the same test (see Figure 8, §B).

We attribute DepFast-Raft's fault tolerance to the use of the DepFast framework to manage potential fail-slow points with the event interface. Specifically, we wrap every set of RPCs with `QuorumEvent` to avoid blocking on a single slow follower, preventing slowness propagation. Besides, blocking operations (e.g., disk I/O) are wrapped and put on a separate thread to avoid blocking the main worker thread.

Note that fail-slow followers inevitably have an impact, as the system is more susceptible to network and disk I/O spikes. In a 3-replica quorum, when one follower fails slow, the quorum reads and writes can be affected by the spikes of the other follower, affecting the tail latency.

6.3 DepFast-Copilot

Figure 6(a) shows the latency and throughput of DepFast-Copilot and the original Copilot implementation [12] in a 3-replica setup. (the 5-replica results can be found in B). The peak throughput of DepFast-Copilot is 33K RPS with a 3-replica setup. The performance is comparable to the original Copilot, which is just 9.3% higher at 36K RPS. The throughput is much higher than Raft, mainly because there is an optimization in Copilot called ping-pong batching that batches many requests into one command.

We inject fail-slow faults to both a leader and a follower, as Copilot is designed to tolerate any one of the fail-slow nodes.

Fail-slow follower. Figure 6(b) and 6(c) show the throughput and latency CDF of DepFast-Copilot with a fail-slow follower. Similar to the results of DepFast-Raft (§6.2), the faults do not have a significant impact on system-wide performance. The differences in throughput are with 10%, and the differences in median latency and P99 latency are within 9% and 30%, respectively. We observe that DepFast-Copilot's tail latency is more susceptible to fail-slow faults than that of DepFast-Raft. Apart from the spikes discussed in §6.2, we attribute that to Copilot's property of having two leaders. With a maximum number of minority followers being slow, a reply from another leader must be obtained to form a quorum, while with fewer slow followers a leader can form a quorum just with replies from non-slow followers. However, the load on a leader is much higher than a follower, causing the tail latency of replies from a leader node to be higher than replies from a follower node. That in turn results in longer tail latency to form a quorum, rendering the tail latency higher.

Fail-slow leader. Figure 6(b) and 6(d) shows the throughput and latency CDF of DepFast-Copilot with a fail-slow leader in a 3-replica setup. With Copilot's multiple-leader design, there is no significant impact on throughput (the differences are within 25%) and the increase in median and P99 latency are within a reasonable range under the existence of one slow leader. For DepFast-Copilot, the CPU contention has the most significant impact on tail latency, in which case the P99 latency increased by 10ms.

The original Copilot implementation. We did the same fault-injection experiments on the original Copilot implementation (results are in Figure 9, §B). We can successfully duplicate the results in its paper that Copilot can tolerate a node slowdown in a slow network simulation. In some other cases (CPU, memory), we find that the original Copilot implementation cannot tolerate the failures as well as DepFast-Copilot. On one

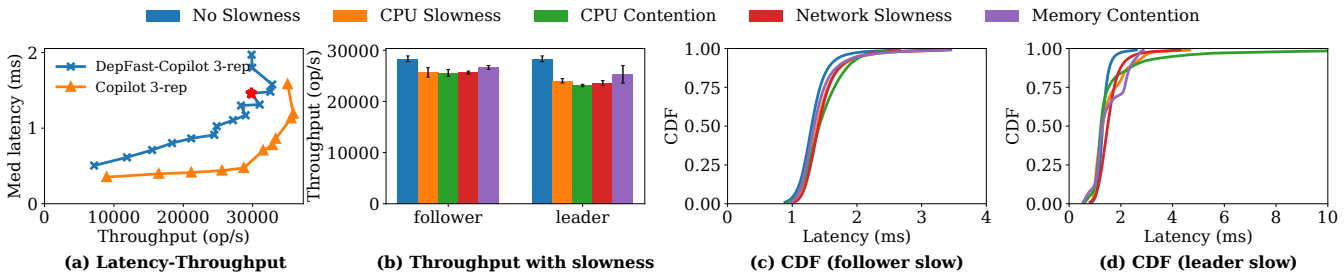


Figure 6: Performance of DepFast-Copilot with various fail-slow faults on follower(s) and leader in the 3-replica deployment.

hand, this is justifiable as the original Copilot implementation is an academic prototype that focuses on verifying the design, rather than the implementation problems. On the other hand, it proves that fault tolerance is not only a design problem but also an implementation problem. It further proves the value of having another layer like DepFast in the system.

7 Related Work

Synchronous versus asynchronous programming. The discussion on synchronous and asynchronous programming styles started decades ago [41]. As common wisdom, synchronous programming (using threads instead of callbacks) is easy to follow, but tends to have unstable performance due to the overhead of OS threads [15,25,26,37,42,43,52,53]. Prior work on synchronous programming focuses on reducing its overhead using cooperative task scheduling with lightweight user-space threads (e.g., coroutines or fibres) [15,43,52,53]. Today, coroutines and cooperative task scheduling have been widely accepted, with built-in support in modern languages such as Go and C++. Our work follows the same principles, and extends the literature by considering distributed systems code (prior work focuses on I/O operations on a single node). An orthogonal direction is to improve the understandability of callback-style code, making it synchronous code alike [25,37]. The approach needs compiler support and extra tooling.

Distributed programming patterns and frameworks. The actor model is a common model to construct distributed programs, e.g., Erlang/OTP [7] and Scala/Akka [1], Orleans [21] and ActOp [46]. Our target is not actor systems, as we focus on the imperative coding style with RPC for communication that is still a common practice in building C/C++ system software. But our results can be complimentary to the actor world, because the actor systems are mostly asynchronous, and may be subject to the same callback hell problem [56].

Rex [32], Eve [35], and Crane [24] target fault tolerance at the OS process level. Ambrosia [30] extends the actor model with built-in fault tolerance support. These frameworks are potential users of DepFast—they can use DepFast to build fault-tolerance mechanisms and services.

A few frameworks help programmers to match their implementations to the specifications rigorously and thoroughly. Mace [36] translates specification into a C++ implementation and provides a model checker to verify correctness. Rules-based programming [50] promotes programming in an event-based state machine, which helps specify concurrent and non-deterministic conditions. Verdi [54] and Ironfleet [33] help build formally verified systems. DepFast has a different goal: making distributed system code easier to write, maintain, and debug. It also addresses different issues, e.g., fail-slow fault propagation and backlogs (reported in formally verified quorum implementations [29]). DepFast also imposes much fewer restrictions on how distributed systems are programmed.

8 Concluding Remarks

We have presented DepFast, a programming framework to build quorum systems. Our experience of using DepFast is encouraging. DepFast helped us to effectively develop high-performance, fault-tolerant quorum systems with complex consensus protocols. With DepFast, we can write quorum systems code that is easy to follow and maintain. Our future work includes using DepFast to build different types of distributed systems, such as sharded datastores with distributed transaction protocols which also have complicated waiting conditions. We will investigate adopting DepFast's abstraction with other frameworks and interfaces, e.g., C++ 20's coroutine interface, and the actor model in Erlang/Scala.

Acknowledgments

We would like to express our deep appreciation to our shepherd, Jon Howell, who was very responsive during our interactions with him and provided us with invaluable suggestions, which have fundamentally improved this paper and strengthened our work. We also thank the anonymous reviewers for their feedback. We thank the authors of Copilot, especially Khiem Ngo, for the discussions and reviews. We thank Dan Plyukhin for discussions that helped us understand the actor programming model deeper. We thank our industry collaborations for the discussions, especially Ye Ji (CockroachDB) and Siyuan Zhou (MongoDB). This work was supported in part by NSF CNS-2130590 and CNS-2130560, and Microsoft Azure credits.

References

[1] Akka. https://www.akka.io/.

[2] braft. https://github.com/baidu/braft.

[3] CMU-efficient/EPaxos, func handlePreAcceptReply. https://github.com/efficient/epaxos/blob/791b115669fca472d3136f6a2eda46c00b3f8251/src/epaxos/epaxos.go#L1000.

[4] CMU-efficient/EPaxos, issue 10. https://github.com/efficient/epaxos/issues/10.

[5] CockroachDB/cockroach pull request 81136. https://github.com/cockroachdb/cockroach/pull/81136.

[6] epoll(7) — linux manual page. https://man7.org/linux/man-pages/man7/epoll.7.html.

[7] Erlang/OTP. https://www.erlang.com/.

[8] etcd. https://etcd.io/.

[9] gRPC. https://grpc.io/.

[10] kqueue(2) - openbsd manual pages. https://man.openbsd.org/kqueue.2.

[11] libev. http://software.schmorp.de/pkg/libev.html.

[12] Princeton-sns/Copilot. https://github.com/princeton-sns/copilot/blob/main/src/copilot/copilot.go.

[13] The Boost Library: Coroutine2. https://github.com/boostorg/coroutine2.

[14] The Go Programming Language. https://go.dev.

[15] 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).

[16] 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)* (April 2018).

[17] BALAKRISHNAN, M., FLINN, J., SHEN, C., DHARAMSHI, M., JAFRI, A., SHI, X., GHOSH, S., HASSAN, H., SAGAR, A., SHI, R., ET AL. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (October 2020).

[18] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Proceedings of the 2000 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'00)* (June 2000).

[19] 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).

[20] 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)* (Seattle, WA, USA, November 2006).

[21] BYKOV, S., GELLER, A., KLIOT, G., LARUS, J. R., PANDYA, R., AND THELIN, J. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)* (October 2011).

[22] CHANG, S. S. Leveraging Go Concurrency. *Go Web Programming, Chapter 9, Manning Publications Co.* (July 2016).

[23] 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)* (Hollywood, CA, USA, 2012).

[24] CUI, H., GU, R., LIU, C., CHEN, T., AND YANG, J. Paxos Made Transparent. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'15)* (October 2015).

[25] CUNNINGHAM, R., AND KOHLER, E. Making Events Less Slippery with eel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS'05)* (June 2005).

[26] DABEK, F., ZELDOVICH, N., KAASHOEK, F., MAZIERES, D., AND MORRIS, R. Event-driven Programming for Robust Software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop (EW 10)* (July 2002).

[27] DO, T., HAO, M., LEESATAPORNWONGSA, T., PATANAANAKE, 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)* (October 2013).

[28] EDWARDS, J. Coherent Reaction. Tech. Rep. MIT-CSAIL-TR-2009-024, Computer Science and Artificial Intelligence Laboratory, June 2009.

[29] FONSECA, P., ZHANG, K., WANG, X., AND KRISHNAMURTHY, A. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the 11th ACM European Conference on Computer Systems (EuroSys'17)* (April 2017).

[30] GOLDSTEIN, J., ABDELHAMID, A., BARNETT, M., BURCKHARDT, S., CHANDRAMOULI, B., GEHRING, D., LEBECK, N., MEIKLEJOHN, C., FAROOQ, U., NEWTON, R., GHOSH, R., ZACCAI, T., ZHANG, I., GOLDSTEIN, J., ABDELHAMID, A., BARNETT, M., CHANDRAMOULI, B., GEHRING, D., LEBECK, N., MEIKLEJOHN, C., MINHAS, U. F., NEWTON, R., PESHAWARIA, R. G., ZACCAI, T., AND ZHANG, I. A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications. *Proceedings of the VLDB Endowment 13*, 5 (January 2020).

[31] 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)* (February 2018).

[32] GUO, Z., HONG, C., YANG, M., ZHOU, D., ZHOU, L., AND ZHUANG, L. Rex: Replication at the Speed of Multi-Core. In *Proceedings of the 9th ACM European Conference in Computer Systems (EuroSys'14)* (April 2014).

[33] 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 ACM Symposium on Operating Systems Principles (SOSP'15)* (October 2015).

[34] 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)* (Boston, MA, June 2010).

[35] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., AND DAHLIN, M. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (October 2012).

[36] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)* (June 2007).

[37] KROHN, M., KOHLER, E., AND KAASHOEK, M. F. Events Can Make Sense. In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX ATC'07)* (June 2007).

[38] LAMPORT, L. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS) 16*, 3 (May 1994).

[39] LAMPORT, L. Paxos Made Simple. *SIGACT News 32*, 4 (December 2001), 18–25.

[40] LAMPORT, L. Fast Paxos. Tech. Rep. MSR-TR-2005-112, Microsoft Research, July 2005.

[41] LAUER, H. C., AND NEEDHAM, R. M. On the Duality of Operating System Structures. *ACM SIGOPS Operating Systems Review (OSR) 13*, 2 (April 1979).

[42] LEE, E. A. The Problem With Threads. *IEEE Computer 39*, 5 (May 2006).

[43] LI, P., AND ZDANCEWIC, S. Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-Level Concurrency Primitives. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)* (June 2007).

[44] LIU, B., LIU, P., LI, Y., TSAI, C.-C., DA SILVA, D., AND HUANG, J. When Threads Meet Events: Efficient and Precise Static Race Detection with Origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)* (June 2021).

[45] 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)* (November 2013).

[46] NEWELL, A., KLIOT, G., MENACHE, I., GOPALAN, A., AKIYAMA, S., AND SILBERSTEIN, M. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)* (2016).

[47] 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)* (November 2020).

[48] 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).

[49] OUSTERHOUT, J. Why Threads Are A Bad Idea (for most purposes). In *Presentation given at the 1996 USENIX Annual Technical Conference (USENIX ATC'96)* (January 1996).

[50] STUTSMAN, R., LEE, C., AND OUSTERHOUT, J. Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)* (July 2015).

[51] 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)* (February 2019).

[52] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)* (October 2003).

[53] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)* (October 2001).

[54] 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).

[55] YOO, A., WANG, Y., SINHA, R., MU, S., AND XU, T. Fail-slow fault tolerance needs programming support. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS'21)* (June 2021).

[56] ZAMORA-GÓMEZ, E., GARCÍA-LÓPEZ, P., AND MONDÉJAR, R. Continuation Complexity: A Callback Hell for Distributed Systems. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing (EuroPar'15)* (August 2015).

[57] 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)* (October 2015).

[58] ZHANG, Q., LIU, V., ZENG, H., AND KRISHNAMURTHY, A. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference (IMC'17)* (November 2017).

[59] 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)* (April 2018).

[60] ZHOU, S., AND MU, S. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)* (April 2021).

A Appendix

A.1 Artifact evaluation

Abstract

A Docker image is provided which contains required dependencies and source code to run the system. Instructions are provided in the `README.md` to reproduce the major results.

Contents

The artifact evaluation includes experiments in Figures 5 and Figures 6. This artifact does not include (1) experiments for comparisons: etcd and ref-copilot, and (2) experiments in B.

Hosting

You can find the publicly available source code at https://github.com/stonysystems/depfast-ae/tree/atc_ae.

Requirements

At least one client plus five servers are used to reproduce the experimental results. Five servers must have an extra disk mounted for slowness experiments. We run all our code on `Debian-10`, which mainly depends on common Linux libraries (i.e., python, gcc and libyaml-cpp-dev). You can install all dependencies by `bash ./dep.sh`.

A.2 Empowered analysis

DepFast empowers a number of analysis to help programmers understand the fault-tolerance properties of their systems and to detect faults at the run time.

Monitoring with linked coroutines Through the event interface, the DepFast framework can link coroutines together and analyze fail-slow fault propagation. For example, the `RpcEvent` could link the caller and the callee coroutines. The framework will propagate the wait-for information and aggregate them at configured granularity. Figure 7 presents an example of a fail-slow fault propagation graph which shows the wait-for relationship at the node granularity in our DepFast-Raft implementation (§5.1). Each vertex represents a node in a quorum. Each edge is directed and weighted: the direction suggests the wait-for relationship; the weight is the count of the waiting. Each edge is colored. A wait on a potential fail-slow event (e.g., an `RpcEvent`) leads to a *red* edge. A wait on a `QuorumEvent` leads to a *green* edge. This graph is generated and refreshed periodically at runtime. It can be used with graph analysis to detect execution paths which are vulnerable to fail-slow fault, that is, the execution path that contains a red edge. Ideally, this graph should not contain any red edges except for those representing a client issuing requests to a server.

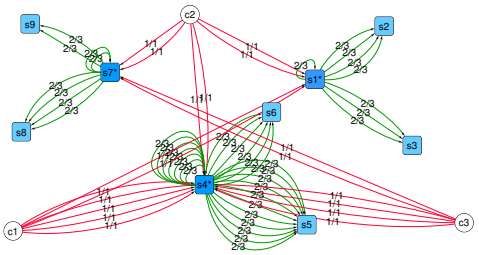


Figure 7: The fail-slow fault propagation graph of DepFast-Raft with three quorums (§5.1). The labels on the edge represents the quorum of the event. "2/3" refers to `QuorumEvent` where 2 responses are needed out of 3 RPCs; "1/1" refers to waiting on a single RPC (clients wait for the leader nodes).

Fault detection. DepFast has a few built-in fail-slow fault detection mechanisms. First, DepFast measures the CPU usage of the worker thread. When the worker thread is awake from `epoll` sleep, it should take up all the CPU core it is running on, because the worker thread does not have thread-blocking calls. The measurement excludes all the `epoll` sleep time and only measures the code executed in the worker thread. If it observes that the worker thread occupies less CPU time than it should, it alerts: either a fault occurs or other programs compete for CPU. Note that this is not perfect detection: the competition could be healthy if it is a shared host; a fail-slow fault could also make the `epoll` sleep longer rather than reducing the CPU utilization.

Second, DepFast measures the time of waiting on each event. If there is a spike for the same event, DepFast will report it. Or, if the wait time repeatedly breaks a user-configured threshold, DepFast will report fail-slow as well.

Third, DepFast exposes the runtime information of resource utilization by allowing applications to register a user-defined detector function. The user-defined detector will be called periodically (taking the monitoring information as inputs) and then make its own decision to notify the application.

B Supplemental evaluation

Figure 8 shows the results of etcd under various fail-slow faults. Our results show that etcd can tolerate these failures well as a production system.

Figure 9(b) and 9(d) shows the throughput and latency CDF of the original Copilot implementation with a fail-slow leader in a 3-replica setup. The experiment verifies that the original Copilot implementation can tolerate a fail-slow node in the cases tested in the Copilot paper. We find that largely due to its immature implementation, in some fail-slow follower cases that are not tested in the original paper, the performance is lower than expected. For example, our experiment of injecting CPU slowness to the original Copilot frequently fails. After some diagnosing, we found that when the follower fails

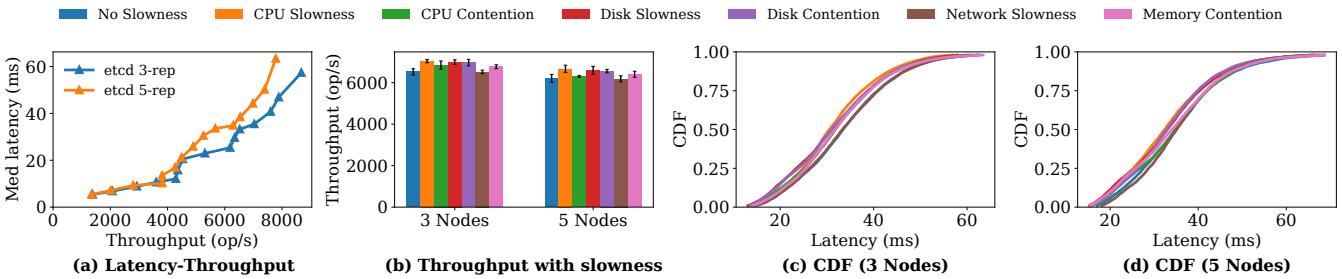


Figure 8: Performance of etcd with various fail-slow faults on follower(s) in the 3- and 5-replica deployments.

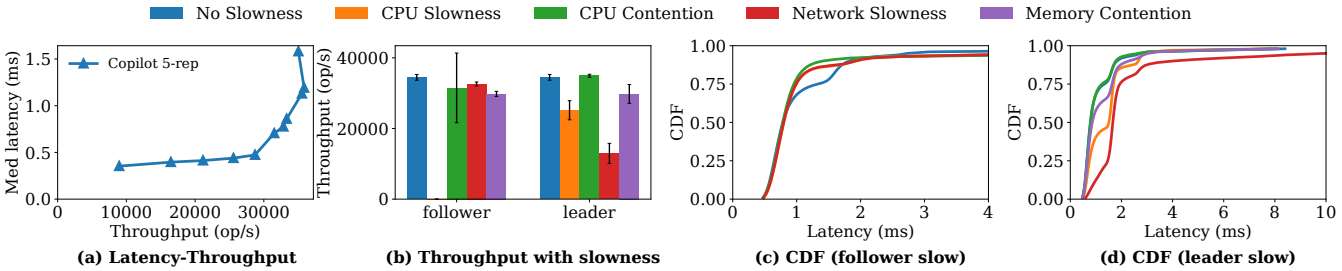


Figure 9: Performance of original Copilot with various fail-slow faults on follower(s) and leader in the 3-replica deployment.

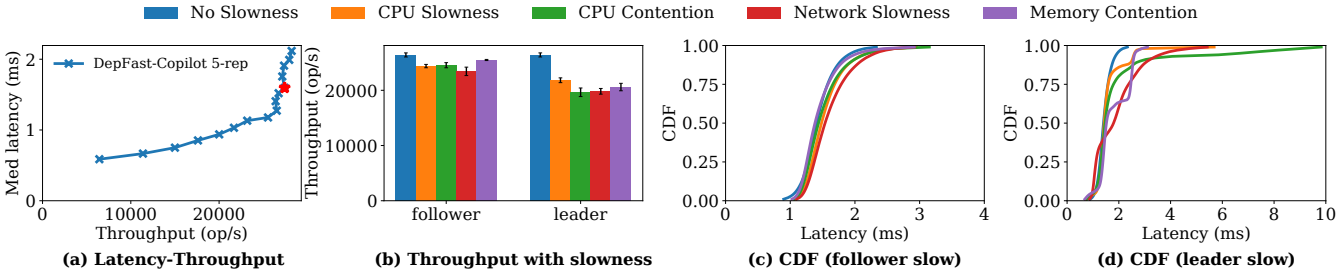


Figure 10: Performance of DepFast-Copilot with various fail-slow faults on follower(s) and leader in a 5-replica setup

slow, it could detect the leader as dead as it cannot process the heartbeats from the leader in time and starts a view change. The leader then steps down and crashes due to what is suspected to be a memory bug. The fail-slow follower will next become the new leader. This results in two live nodes, with one of them failing slow, rendering the cluster into a slow or even stalled state.

Figure 10 shows the evaluation results of DepFast-Copilot in a 5-replica setup, in a similar vein as Figure 6. Figure 10(a) shows the latency-throughput of DepFast-Copilot and the original Copilot implementation. The peak throughput of DepFast-Copilot is 18% higher than the original Copilot.

Figure 10(b)-(d) show the throughput as well as latency CDF of DepFast-Copilot with fail-slow faults injected to follower and leader nodes. Recall from §6.1 that we inject faults on two followers (denoted as "*follower*" in Figure 10) and on one leader and one follow (denoted as "*leader*" in Figure 10). Similar to the results of 3-replica setup, there is no significant downgrade on throughput and latency in terms of both fail-slow follower and leader. For follower slowness, the decrease in throughput is within 12%. The increase in median and P99 latency are within 10% and 35%, respectively. For leader slowness, the decrease in throughput is within 26%. The latency results are similar to those in a 3-replica setup.

Compared with the 3-replica setup (Figure 6), we find that the 5-replica setup is affected by network and disk I/O spikes more. We discussed the impact of the spikes in §6.2; we elaborate more here. Since we inject one fail-slow node in the 3-replica setup and two fail-slow nodes in the 5-replica setup, the probability of one of the remaining replicas experiencing network or I/O spike is higher in the 5-replica setup than in the 3-replica setup. As every request leads to a quorum read or write, the 5-replica setup is more susceptible to network or I/O spikes. The spikes, in our experience, are common in Azure Cloud (also reported by other studies [16, 51, 58, 59]). Note that Azure uses virtual hard drives that are accessed remotely over the network [59]. As a result, disk writes can also be impacted by network spikes.