Eventually Durable State Machines

by

Kriti Kathuria

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Consider a key-value store. Typically, it is designed to ensure the data it stores is durable. The key-value store guarantees that client updates will persist, regardless of failures. Durability is achieved by replicating the updates to multiple machines. This replication adds to the time it takes for the key-value store to respond to the client request. For latency-sensitive use cases, the key-value store may offer to acknowledge client requests without awaiting full replication.

The Eventually Durable "ED" State Machine extends the state machine model of building applications [25] and allows applications like the above key-value store to make latency/durability tradeoffs. The ED model allows applications to respond faster but leaves them vulnerable to data loss in case of failures. Clients of ED applications must accept that the application might lose some updates in the event of a failure. The ED model presents clear failure semantics for the client to reason about the lost updates. Further, it also provides tools for the clients to manage the risk associated with a weak durability guarantee.

Continuing the key-value store example, writes to an ED key-value store are not guaranteed to be durable when they return. Writes may become durable eventually, after a response has been sent. The client receives a response and proceeds under the assumption that the write will eventually become durable. Thus, the client speculates on the eventual durability of the writes it makes to the ED key-value store.

Failures may lead to the loss of acknowledged writes. The ED model guarantees that if a write is lost in a failure, all writes after it will be lost as well. Furthermore, writes that are lost are lost permanently. The ED model also guarantees that writes that survive the failure, survive forever. In this way, failures result in the resolution of existing speculation and there will only ever be a single speculative "future" for the ED application.

We develop ED Raft, a variant of the Raft consensus algorithm [22], to implement the ED state machine. ED Raft acknowledges client proposals early, without waiting for them to fully replicate. We present the ED Raft protocol and its key properties. Further, we show that ED Raft supports the ED failure model described above. Finally, we describe the implementation of ED Raft as a wrapper on top of an existing Raft implementation.

Acknowledgements

I believe achieving any major goal is less a function of doing things and more a function of identity, of becoming. Major goals are seldom simply items to check off, and almost always, cause huge shifts in ways of thinking and ways of being. And therefore, major goals are so profoundly transformative that there is a clear before and after. I am a vastly different person leaving Waterloo than who entered Waterloo. I would like to acknowledge everyone who has played a role in this evolution.

The work of a teacher is the work of God. I am infinitely grateful to my supervisor Dr. Ken Salem. For his guidance: this thesis would not be what it is without his ideas and expertise. For his generosity with time and attention and always being available whenever I needed help. For his unending patience and understanding, without which I would not have been able to learn the things I did. It is all thanks to him that I was able to experiment and explore as freely as I did.

Not everything can be taught. Somethings can only be learned by proximity and observation. I consider myself fortunate to have spent so much time around Professor Ken, to have been able to build an intuition for research, for the right kind of questions and the right level of detail. I will always be grateful for his unending patience as I crafted my mental models.

Our ideas are nothing without our ability to communicate them. I have been able to develop a good sense for effective communication and presentation and it is all owing to his feedback on countless (very bad!) drafts and dry runs and also all the one on one meetings, which provided feedback of a different kind.

I shall forever be indebted for everything I have learned from him.

Committee members Dr. M. Tamer Özsu and Dr. Sujaya Maiyya, for reading my thesis and providing valuable feedback and insight to better contextualise this research.

UWaterloo, for providing a space for pure academic pursuit, to learn and grow. The school and campus, especially DC, will forever hold a special place in my heart.

My friends, for the fun and laughter, for supporting me when the going got tough, and for conversations that provided at times, shared and at times, different perspectives. I am a more grounded person because of them. I am also deeply grateful to them for finding the time to be present for my thesis presentation.

My colleagues at Waterloo and CWI, whom I got to learn from and emulate.

I would be remiss not to mention SCS, the CS grad office, and CSCF for never leaving me lacking for any assistance or resources as I navigated my graduate career.

The places we have been inform our present as much as the place we are in. And so I would like to mention my past mentors, and the institutions and organisations I have been a part of. Those experiences shaped my time at Waterloo as much as all people mentioned above.

My sister, for being a constant source of inspiration, who has asked me to mention that she is 6 years younger.

My parents, everything I am, I owe to them.

Dedication

for the dreamers...

Table of Contents

List of Figures

Chapter 1

Introduction

In this chapter, we make the case for Eventual Durability and present an overview of the ideas and approaches discussed in this work.

We start by discussing the problem statement and introduce Eventual Durability. Then, we justify the problem statement by presenting the results of an initial experiment. Finally, we summarize our contributions and present an outline for the rest of this work.

1.1 The Case for Eventual Durability

Picture yourself grocery shopping online. You are rushing through the process, mentally checking off items on your list. You swiftly click on the items you need, one eye on the cart, making sure the item appears in the cart. You don't want items to be missed. You are expecting the item to promptly appear in the cart, but it doesn't. Instead, a loader has appeared and keeps going for one second, two seconds, three, four... You are now annoyed. You have ten other chores to get to and the online shop is taking forever to do the simple task of adding an item to the cart!

Speed is a critical aspect of good user experience and in today's world, it is a deciding factor in user retention. And so, applications care a lot about the time it takes to respond to user requests and take extensive measures to reduce it. One reason for this latency is that user data is made *durable* before the application responds to user actions. The application acknowledges any user updates only once it has ensured the update will not be lost. Guaranteeing durability requires time and adds to the user response latency.

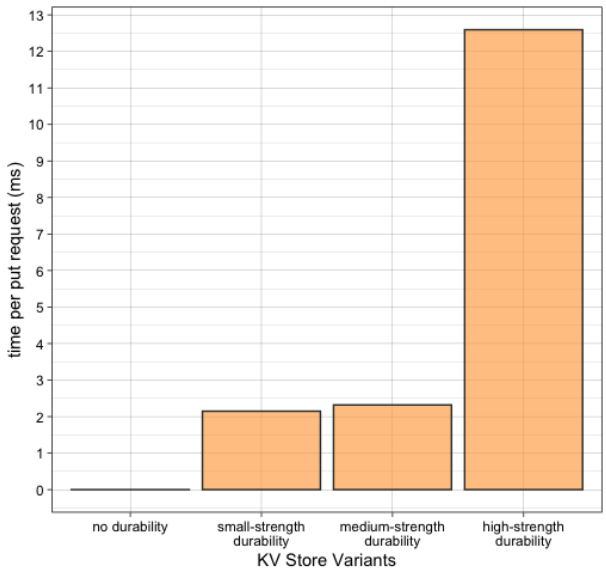

Figure 1.1: Cost of Durability.

The cost of durability is significant at a minimum of 2 ms per write. Most applications opt for medium-strength durability.

So applications implement ad-hoc mechanisms to get around the cost of durability, such as acknowledging the user without waiting for the request to become durable. In other cases, applications may appear to have performed the desired action, while executing requests in the background, retrying as needed. These methods are informal and context-dependent, and failures and deviations from expected behavior are handled on an as-they-come basis. Moreover, the application has to assume the burden of reasoning about data loss and consistency when failures occur, which given the complexity of modern-day applications, is not a small undertaking.

In this work, we propose Eventual Durability, a principled approach to make durability/latency tradeoffs. The ED model presents clear failure semantics so that applications can reason about the data loss that can occur in such tradeoffs.

1.2 Quantifying the Cost of Durability

In the previous section, we declared that durability is expensive for the application. Now, we make the cost of durability concrete. We begin by discussing why durability is expensive.

Durability is the guarantee that user data will never be lost. Typically, durability is achieved by *replication*. When the application receives a user request, it makes copies of the request on multiple machines before acknowledging the request. These machines may be geographically separated from the main application server and the application may have to perform network calls to create copies. These network calls are time-consuming and the reason why durability is expensive.

The farther apart the machines, the stronger the durability guarantee, and the costlier the durability guarantee. For example, if all the machines are present in the same data center, copy creation will happen over the faster, local area network. However, the application will not survive data center outages. If the machines are present in different data centers, copy creation will require the slower, wide-area network calls. On the upside, the application will survive data center outages.

We compare the response times of four key-value stores. One of these is a "no-durability" key-value store that is a simple single-node, in-memory hash table and provides no durability guarantee. The other three key-value stores guarantee durability and respond only after making a client request durable. These three "durability-guaranteed" key-value stores vary in durability "strength".

Each of the durability-guaranteed key-value stores is a cluster of three machines that respond after ensuring that at least two machines in the cluster have a copy of each client request. The difference between them is the distance between the machines: the first has all three machines in the same data center and offers "small-strength durability", the second "medium-strength durability" places machines in different data centers that are geographically close together, and the third "high-strength durability" spreads the machines across large geographic distances. In the real world, applications opt for at least medium-strength durability. More details about the experimental setup can be found in §4.3.

Fig.1.1 depicts the average time to perform 10,000 inserts into each key-value store variant. Inserts into the no-durability variant are instantaneous, while inserts into the durable variants take at least 2 ms. Most applications prefer medium-strength durability, which adds 2.5 ms to every insert. This is the cost of durability and it further increases for a geo-replicated cluster.

1.3 Research Contributions

As described above, applications are typically designed to ensure that the data they store is durable. One of the ways to achieve durability is to make copies of the data on multiple machines. This is especially true in the modern cloud context where data is not durable unless redundant copies exist. When an application receives a client request, it replicates the client request to multiple machines and then responds to the client. This replication introduces a delay in responding to the client and so for latency-sensitive use cases, the application may circumvent replication costs by responding to clients without awaiting full replication.

We propose the Eventually Durable State Machine, a principled approach to allow applications to make durability/latency tradeoffs. The EDSM extends the state machine model of building distributed applications[25]. The state machine approach is an approach to implement replication where an application has multiple replicas, and a client request is saved on at least a quorum of replicas before a response is sent. When one replica fails, another can take its place.

The ED state machine also consists of multiple replicas and is structured like the classical state machine. However, unlike the classical state machine, the ED state machine responds *without* waiting for replication to complete. Failures may lead to data loss as the new replica may not have the client requests the failed replica acknowledged. Clients of the ED state machine accept the risk of data loss. In exchange, the ED state machine offers fast response time and presents clear failure semantics for clients to reason about data loss.

The ED state machine doesn't guarantee the durability of writes, unlike the classical state machine which guarantees write durability. Eventually, writes to the ED state machine may become durable. The client receives a fast response and proceeds under the assumption that the write will eventually become durable. In this way, the client speculates on the eventual durability of the writes it makes to the ED state machine.

In the absence of failures, ED applications behave like the classical, non-ED applications. But if failures occur, some acknowledged writes may be lost as they may not have become durable at the time of failures. The ED model guarantees that if a write is lost, all writes after it will also be lost. Further, the lost writes are lost permanently and never reappear, and the writes that survive the failure become durable. Aside from the standard read and write operations, the ED model offers a *sync* operation that ensures the durability of preceding write operations. sync is a tool to resolve all speculation and allows clients to manage the risk associated with durability speculation.

Thus, applications based on the ED state machine are fast but are also vulnerable to data loss. We argue that applications already make such tradeoffs. The ED model just formalizes this notion and presents a clear failure model to reason about the possible data loss.

We present a log abstraction called the ED Log, which can be used to implement the replication for the ED state machine. We describe the evolution of the ED Log over time as the state machine processes client requests, and across failures.

We develop ED Raft to implement the ED Log. ED Raft is the eventually durable variant of the Raft consensus algorithm. Essentially, ED Raft does not wait for client requests to replicate before acknowledging them: client requests become durable *eventually* after they have been acknowledged. Failures may lead to the loss of some acknowledged writes. ED Raft guarantees that the data loss always falls within the boundary established by the ED model.

As ED Raft is an eventually durable variant of Raft, we base our implementation on an existing implementation of Raft. We choose etcd Raft [1] as our base Raft implementation. etcd Raft is structured like a library. Applications import the library and call it with updates they want replicated. The library returns once the updates are replicated and are durable.

We implement ED Raft as a wrapper around the etcd Raft library. With this design, applications using etcd Raft can adopt the ED model simply by installing the wrapper. We evaluate an eventually durable key-value store backed by ED Raft and show that it is at least 2x faster than a key-value store that guarantees durability.

In §2, we introduce the Eventually Durable State Machine and define its failure model. We contrast it with the classical state machine to aid understanding. In §3, we present ED Raft and its key properties, and establish that a state machine backed by ED Raft behaves like the ED state machine. Appendix A contains ED Raft's formal correctness proof. In §4, we describe the implementation of ED Raft and present our evaluation. In §5, we present a discussion of related works. Finally, we conclude and discuss the future directions of this research in §6.

Chapter 2

The Eventually Durable State Machine

In this chapter, we introduce the Eventually Durable State Machine and provide an abstract specification. It extends the state-machine approach to building fault-tolerant distributed applications[25]. The classical state-machine approach allows applications to offer a strong durability guarantee: effects of user operations persist despite failures. Durability is an expensive guarantee. Eventually Durable state machine is a principled approach to weaken the durability guarantee in return for faster operations.

We start by introducing the classical state machine approach in §2.1. Then, in §2.2, we describe the eventually durable state machine and its guarantees. We formally define the eventually durable state machine in §2.3. Finally, in §2.4 we consider the implications of eventually durable state machines on clients.

2.1 The State Machine Approach

The state machine approach describes the state machine as consisting of a set of state variables and a set of operations, typically read and write, that modify the state variables and/or produce an output. Clients perform operations on the state machine. These operations permanently modify the state such that the changes persist across system failure. In other words, they are durable. The state variables, along with their values, encode the state of the state machine.

Most applications can be modeled as a state machine. Take a key-value store as an example. A key-value store is a state machine that stores a collection of key-value pairs. Clients can perform a write operation *put(key, value)* or a read operation *get(key)*. *put* creates or updates the key-value pair. *get* returns the value associated with the key. *put* requests are durable. This means that updates to the key-value store are not lost, and the client will always be able to read the writes it performs. The state is all the key-value pairs present in the key-value store.

2.2 The Eventually Durable State Machine

The **Eventually Durable "ED" state machine** extends the state machine approach to allow applications to weaken the durability guarantee in return for faster operations.

In the ED model, writes are not durable when they return, unlike the classical model in which writes are durable when they return. Writes may become durable eventually, sometime after returning. This means that the acknowledged writes may be lost when failures occur, causing the application to revert to an older state. However, in the absence of failures, an ED application behaves like the standard, strongly durable application with faster response times.

The client, on its part, receives a fast response and proceeds, speculating on the eventual durability of the requests it has issued. ED applications allow clients to perform a special *sync* operation. sync forces the durability of all preceding requests and is a tool for the clients to manage the risk associated with durability speculation.

Consider an ED key-value store. *put* requests to the ED key-value store are not guaranteed to be durable when they return. They may become durable eventually or may be lost if a failure occurs, causing the client to see an outdated value of a key. *sync* makes all preceding *put* requests durable. Future reads will not observe a version of the key older than when *sync* was called.

ED failure model. The ED model provides clear failure semantics to reason about the lost client requests. The ED model guarantees that if a write is lost in a failure, all writes after that will be lost as well. The speculative writes that survive a failure will become durable and will survive forever, and the writes that are lost are lost permanently and will never reappear. In this way, a failure results in the resolution of all existing speculation and there will only ever be a single speculative "future" for the state machine.

The below example illustrates the behavior of the ED state machine.

Example 1. The following shows a sequence of write requests w_i issued by the client, and the state s_i after each request:

$$[\ ^{s_0}\ \ w_1\ ^{s_1}\ \ w_2\ ^{s_2}\ \ w_3\ ^{s_3}\ ]$$

Classical read following w_3 will always return s_3. In an ED state machine, the same read may return s_3. However, it may also return s_0, s_1, or s_2.

If the read returns a state older than s_3, it is because w_3 has been lost in a failure. Writes are lost permanently and do not reappear. Thus, if the read does not return s_3, then no future read will return s_3 either. If a read returns a stale state, the next read, barring any new writes, will always return the same state or a state staler than the one the first read returned.

Let us assume that w_3 is lost and then the client issues a sync and then another write.

$$[\ ^{s_0}\ \ w_1\ ^{s_1}\ \ w_2\ ^{s_2}\ \ \cancel{w_3}^{\cancel{s_3}}\ \ sync\ ^{s_2}\ \ w_4\ ^{s_4}\ ]$$

A read at this stage may return s_4 or s_2, but will never return a state older than s_2, because writes up to w_2 are guaranteed to be durable once the sync operation has returned. A read does not return a state older than the latest durable state.

2.3 Definitions

We now formally define the Eventually Durable State Machine. We start with the definition of the classical state machine and place it in contrast with the ED state machine.

Definition 1. ***Classical State Machine****. A classical state machine over state space $\mathbb{S}$ and having initial state $I \in \mathbb{S}$ consists of a current state $S \in \mathbb{S}$, and supports read and write operations.*

A read operation returns the state of the state machine at the given time. A write operation moves the state machine from one state S to another state S', $\in \mathbb{S}$. We represent the new state S' produced by applying a write operation w to S as $w(S)$. Thus, $S' = w(S)$.

The current state is represented using D. Initially, $D = I$. A read returns D and a write w_i sets $D = w_i(D)$.

Example 2. The following shows a sequence of write requests issued by the client to the classical state machine, and D and V after each request.

$$\left[\ {}^{D=I}\quad w_1\ {}^{D=w_1(D)}\quad w_2\ {}^{D=w_2(D)}\quad w_3\ {}^{D=w_2(D)}\ \right]$$

A read will always return D. After w_3, $D = w_3(w_2(w_1(I)))$. Thus, the client always sees the effects of all preceding writes. D can be thought of as the durable state.

Definition 2. ***Eventually Durable State Machine****. The eventually durable state machine over state space $\mathbb{S}$ and having initial state $I \in \mathbb{S}$ consists of a current state $S \in \mathbb{S}$, and supports read, write and sync operations.*

A read operation returns the state of the state machine at the given time. A write operation moves the state machine from one state S to another state S', $\in \mathbb{S}$. We represent the new state S' produced by applying a write operation w to S as $w(S)$. Thus, $S' = w(S)$. sync is a special write operation.

The current state is represented using D and a sequence of write operations $V = (w_i, w_j, w_k ... w_n ... w_{n'})$. Initially, $D = I$ and $V = \phi$.

A read sets $V = \langle V \rangle$ and returns $V(D)$. Here, $\langle V \rangle$ represents a non-deterministically chosen prefix of the sequence of writes V, and $V(D)$ represents applying the sequence of writes represented by V to the state D. A write w_i sets $V = \langle V \rangle + w_i$. A sync sets $D = \langle V \rangle(D)$ and $V = \phi$.

Example 3. The tree in Fig.2.1 shows all the possible states of an ED state machine given a sequence of requests. While the tree shows all possible outcomes for the given sequence of operations, only a single path will be followed down the tree.

There are two possible states after w_2: $V = (w_1, w_2)$ and $V = (w_2)$. The first possibility has $\langle V \rangle = \phi$, indicating no writes were lost, and the second has $\langle V \rangle = (w_2)$, indicating the loss of w_1. Similarly, the possible states after w_3 are shown.

V can be thought of as the sequence of speculative operations. Then, $\langle V \rangle$ represents the prefix of speculative operations that survive after a failure.

Level three of the tree in Fig.2.1 depicts possible states after w_3. The second branch depicts the scenario where a failure occurs after w_2 in which w_2 is lost, and then w_3 is issued. The volatile state V becomes $\langle V \rangle + w_3 = (w1, w3)$. A read after this stage may return any of the following: $w_3(w_1(D))$, $w_1(D)$ or D, and subsequent requests will modify the volatile sequence $V = (w_1, w_3)$ or $V = (w_1)$ or $V = \phi$.

D can be thought of as the durable state. The initial state I is always the durable state. *sync* applies a prefix of V to D. The prefix may or may not be the complete V. Subsequent reads will not return a state older than D.

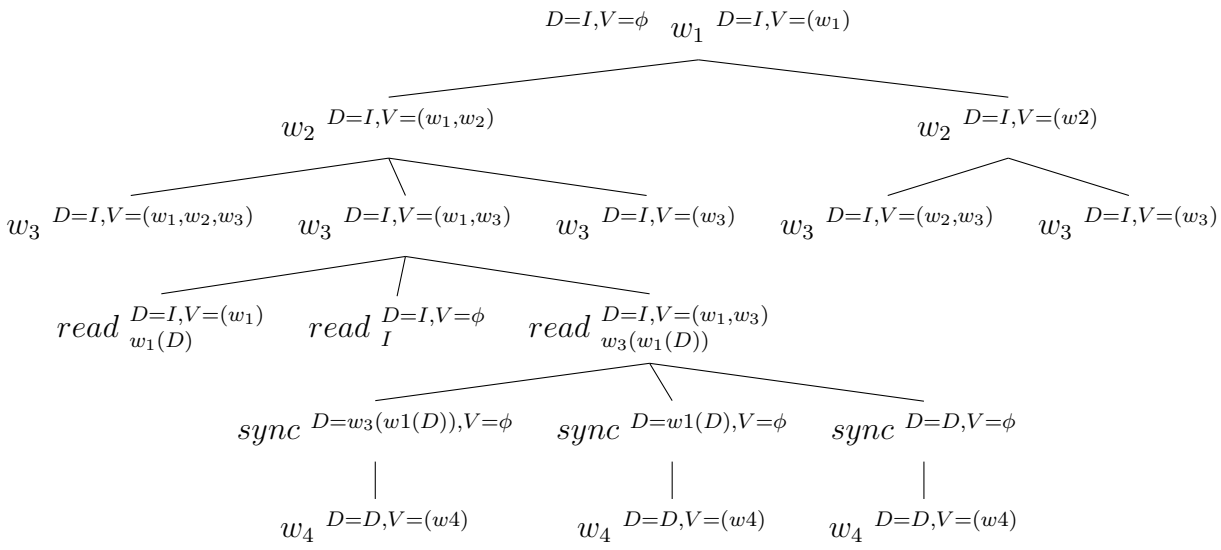

Figure 2.1: Possible states of an ED state machine

2.4 Application Consideration

Classical state machines guarantee durability. The client can reasonably expect that the operations it performs will persist, regardless of failures. For example, if the client sets $a = 2$ and then sets $a = a + 1$, the client can expect that the value of a is 3. In this way, the classical model is deterministic.

The ED model, on the other hand, does not guarantee durability: operations are not guaranteed to be durable unless the client explicitly calls *sync*. Because of this, failures lead to loss of updates, causing the state machine to revert to an older state. This essentially means that the state machine can end up in a state very different from what the client expects, and the client finds out about it only when it performs a read operation.

If the client performs $a = 2$ and then $a = a + 1$ in the ED model, the second write may succeed or fail because the first write is lost. A read after the second write, assuming the write succeeds, can return $a = 3$ if no failure occurs, or $a = 2$ if the second write is lost, or fail altogether if both updates are lost. In this way, the ED model is non-deterministic.

The ED model offers faster client response. However, there is no magic pill and the tradeoff is durability. The ED model increases the complexity of application development. Write durability is achieved in a single operation in the classical model but takes three in

the ED model: a write, a sync, and a read to ensure no data loss has occurred between the write and the sync.

We argue that latency-sensitive applications forgo durability guarantees for performance and they do so in an ad-hoc manner. The Eventually Durable State Machine establishes a principled approach and a strong failure guarantees to allow the applications to reason about performance/durability tradeoffs they already make.

Further, we argue that in a majority of cases, writes will become durable. Writes are lost when failures occur and in the context of eventually durable state machines, failures occur when the application server fails. Typically, the mean time between failures for a single server is several months or more[22]. In the absence of failures, an ED state machine behaves like the classical state machine and guarantees write durability.

Chapter 3

ED Raft

In this chapter, we describe the ED Raft protocol to implement the ED state machine. ED Raft is a variant of the Raft, a protocol to implement the classical state machine [22]. In the state machine approach, durability is achieved by *replicating* a write request to multiple machines before responding. Raft executes client requests after they have been replicated. ED Raft does not await full replication before executing client requests.

Logs are a common way to implement replication in state machines. We define a log abstraction called ED Log such that any application that implements the ED Log behaves like the ED state machine. In §3.1, we introduce the ED Log and describe its behavior.

We hang the development of ED Raft on the ED Log. ED Raft is presented in §3.2. We start by describing the semantics of an ED Log managed by ED Raft and present ED Raft's Safety Property as a function of the ED Log. Then we describe the ED Raft protocol and justify that the Safety remains invariant. Finally, we present ED Raft's Speculation Guarantee and justify that ED Raft supports the ED failure model.

In §3.3 we use examples to describe the operation of ED Raft and highlight its properties.

3.1 ED Log

Replication in state machines is typically implemented using logs. A state machine consists of multiple replicas, a write is executed by these multiple replicas before the state machine responds to the client. Logs are used to ensure that all replicas execute the same sequence

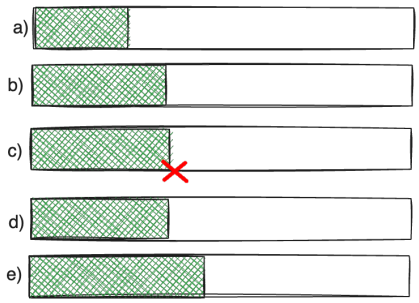

Figure 3.1: Evolution of the classical log

of write requests. An entry for each write is created in all the logs and replicas apply entries from their log to their local instance of the state machine.

Consider an abstract replicated log that represents all client requests processed by the state machine in order. We start by describing the replicated log for a classical state machine. We then develop the ED Log, a replicated log for the ED state machine, and contrast it with the classical replicated log. We describe the behavior of the ED Log over time as the ED state machine processes client requests and across failures.

Classical Log

The classical model guarantees write durability. In the classical model, a write is applied to the state machine only after it has been replicated and become durable. Only writes that are durable are considered to be part of the replicated log.

Fig.3.1 depicts the replicated log for a classical state machine. Each entry in the log represents a write request applied to the state machine. Green indicates that the entry is durable. As the classical state machine applies only durable entries, all entries in the log are green. As time goes on and the state machine processes more write requests, the replicated log grows(b).

In the classical model, writes are not lost in failures. Fig.3.1(d) depicts the classical log after a failure(c). Since writes are durable, the post-failure log is the same as the pre-failure log. The log grows as the state machine processes more writes and more entries are added to the log(e).

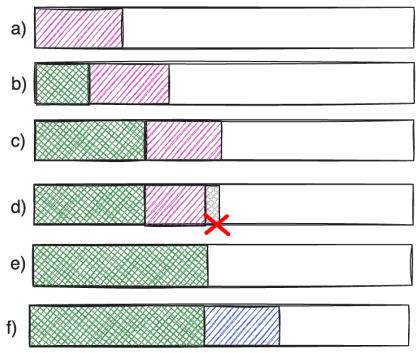

Figure 3.2: Evolution of the ED Log

ED Log

ED state machines do not wait for writes to become durable before applying them. In other words, ED state machines apply writes speculatively. Thus, the replicated log for the ED state machine contains entries that are not yet durable. Over time, these entries become durable.

Fig.3.2 depicts the replicated log for an ED state machine. Pink indicates that the entry is speculative, and as before, green means the entry is durable. At first, all entries in the log are speculative(a). Eventually, as writes get replicated, the speculative entries become durable. This is shown in (b): a prefix of the speculative entries from (a) have become durable or "green". As time goes on and the state machine processes more write requests, more pink entries are added to the log, and more of the speculative entries become green(c).

Since writes may not be durable when the ED state machine applies them, they may be lost in failures. Fig.3.2(d) depicts a failure in which a suffix of the speculative entries is lost (in gray). In the ED model, failures result in a resolution of all existing speculation. (e) depicts the post-failure log: the surviving pre-failure speculation has become durable. As time goes on and the state machine processes more writes, new speculation, represented in blue, is added(f).

3.2 ED Raft

We now present ED Raft. ED Raft is a log-based protocol to implement the ED state machine. The replicated log managed by ED Raft behaves like the ED Log. Therefore, an

ED Raft-backed state machine presents ED semantics.

We start by describing the semantics of an ED Log managed by ED Raft in §3.2.1. Next, we describe ED Raft's Safety Property as a function of the ED Log in §3.2.2. We present the ED Raft protocol in §3.2.2 and show that the Safety remains invariant under the protocol. Finally, in §3.2.4, we describe ED Raft's Speculation Guarantee and use it to justify that ED Raft supports the ED Log behaviour.

3.2.1 ED Raft Log

In this section, we describe the semantics of an ED Log managed by ED Raft. ED Raft is the eventually durable variant of Raft. Therefore, we begin by describing some Raft preliminaries and define the replicated log managed by Raft. We build on top of this and describe the ED Log managed by ED Raft.

Raft Log

Raft is a protocol to implement the classical state machine. In Raft, one of the replicas is elected leader and manages all client requests. The leader creates an entry in its log for each request and replicates it by adding it to the logs of followers. Once the entry is replicated, it becomes durable or in Raft terms, *committed*. After the entry is committed, the leader applies the entry to its copy of the state machine and responds to the client.

A replicated log managed by Raft behaves like the classical log described previously. The green portion of the logs in Fig.3.1 can be seen as the set of committed entries as time progresses.

Further, Raft divides time into *terms*. The set of entries that are durable in a given term t are committed at t or $committed(t)$. For example, for some term t_i, the green portion in Fig.3.1(a) can be thought of as the set of $committed(t_i)$ entries. As time progresses and Raft serves more requests, $committed(t_i)$ grows as in Fig.3.1(b).

When there is a failure, term t_i ends, and a new term t_j starts. A new leader is elected for the new term t_j. All the entries in $committed(t_i)$ are present in $committed(t_j)$, as well as all the new entries committed in t_j. All the entries in $committed(t_j)$ as time evolves can be represented by the post-failure logs in Fig.3.1. The set of committed entries only grows.

Thus, a replicated log managed by Raft contains all the committed entries.

ED Raft Log

ED Raft deviates from Raft in how it applies entries to the state machine. Raft commits an entry and then applies it. ED Raft does not wait for entries to commit and applies them when they are speculative.

The replicated log managed by ED Raft is described by the Logical Log.

Definition 3. *The **logical log** for a given term t consists of all the committed(t) entries, followed by all the speculative entries added by the confirmed leader of term t.*

For some term t, suppose that $C, C \geq 0$ is the number of entries in *committed*(t), and $S, S \geq 0$ is the number of speculative entries added by the confirmed leader of term t. In ED Raft, a leader is *confirmed* if at least one entry added in that leader's term is committed. Then, the logical log for t has C committed entries in positions $[1...C]$ and the S speculative entries in positions $[C + 1...C + S]$.

Fig.3.2(b) can be thought of as the logical log for a term t_i. The green entries are all the *committed*(t_i) entries and the pink entries are the speculative entries added by the confirmed leader of t_i. When there is a failure, t_i ends, and t_j starts. Fig.3.2(e) & Fig.3.2(f) can be thought of as the logical log of term t_j after t_i: blue entries represent the new speculative entries added by the confirmed leader of t_j.

3.2.2 ED Raft Safety

ED Raft's Safety Property constrains the entries replicas can apply to their state machines such that ED semantics are maintained. In ED Raft, replicas apply the log entries that are present in the Logical Log. Thus, ED Raft's Safety Property is as follows.

Property 1. ***State Machine Safety**. A server's state will always reflect a prefix of the logical log for the server's term.*

In other words, servers apply log entries that are present in the logical log of the server's current term. If Fig.3.2(b) represents the logical log for some term t_i, then the state machine of a server at t_i will apply entries from the given log in the figure, and so its state will reflect a prefix of the given log. When the server's term changes to t_j, its state would reflect a prefix of the logical log of the new term.

The state of this server cannot contain the pink entries as it is speculation added by the confirmed leader of another term. Since the server was previously at term t_i and may have applied the pink entries, the pink entries are "un-applied" from its state.

ED Raft guarantees that the Safety Property holds at all times.

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied*	index of highest log entry applied to state machine (initialized to 0, increases monotonically)
state	the state after applying log[lastApplied] (initialized to *initial state*)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries; also used as heartbeat.

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it. Append any new entries not already in the log
4. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

*Although lastApplied is listed as volatile state, it should be as volatile as the state machine. If the state machine is volatile, lastApplied should be volatile. If the state machine is persistent, lastApplied should be just as persistent. [https://github.com/ongardie/dissertation/#chapter-3-basic-raft-algorithm]

RequestVote RPC

Invoked by candidates to gather votes.

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry
lastLogTerm	term of candidate's last log entry

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine.
- If lastApplied >= commitIndex & log[commitIndex].term = currentTerm: increment lastApplied, apply log[lastApplied] to state machine. (apply speculative entries)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower.
- When currentTerm changes, set the state to *initial state* and set lastApplied to 0. (*un-apply* lost speculation)

Followers:

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates:

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPC (heartbeat) to each server; repeat during idle periods to prevent election timeouts.
- If command received from client: append entry to local log, respond after entry applied to state machine.
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N.

3.2.3 ED Raft Protocol

The ED Raft protocol is presented in the figure on the previous page. The figure is adapted from Ongaro et al. (Fig.2, [22]). ED Raft is identical to Raft, except for the revisions shown in green.

ED Raft deviates from Raft in only how entries are applied to the state. In ED Raft, speculative entries are applied if they are part of the logical log for the current term. Additionally, since speculative entries may be lost, ED Raft must "un-apply" these entries from the state. We expound on both below and argue that the Safety Property always holds.

Applying speculative entries. ED Raft allows servers to apply speculative entries if the entry at the server's commit index belongs to the server's current term.

A server's commit index identifies the entries in the server's log that are known to be committed. All entries up to the commit index in the server's log are committed. If the entry at the commit index has the same term as the server's current term, then the leader that created that entry, i.e., the leader of the server's current term is confirmed. Therefore, all entries after the commit index will be speculative entries added by the confirmed leader.

Thus, when a server applies entries, they are either committed entries or speculative entries added by the confirmed leader. And so, the server's state corresponds to a prefix of the logical log.

***Un-applying* lost speculative entries**. In ED Raft, entries are applied speculatively. If these entries are lost, they must be un-applied. ED Raft requires that servers reset their state to *initial state* when their current term changes. This has the effect of removing any speculative entries from the server's state. Trivially, an empty state corresponds to a prefix of the logical log.

Eventually, servers re-apply entries up to their commit index. Servers apply entries after the commit index, i.e., the speculative entries if there is a confirmed leader.

We conclude that the ED Raft's Safety property remains invariant under the ED Raft protocol. The formal correctness proof of the Safety Property is present in Appendix A.3.

3.2.4 Speculation Guarantee

In this section, we justify that ED Raft supports the ED Log behavior, particularly that entries lost in a failure, are lost permanently and entries that survive the failure become

durable. We start by stating ED Raft's Speculation Guarantee, which captures the behavior of the speculative entries across failures.

Property 2. ***Speculation Guarantee****. A speculative entry added in a given term will either be present in the logs of confirmed leaders for all higher-numbered terms or will not be present in the logs of confirmed leaders for all higher-numbered terms.*

We consider the logs of confirmed leaders of two consecutive terms t and t'. The log of the confirmed leader for t contains all the $committed(t)$ entries and speculation added by the confirmed leader. Fig.3.2(b) and Fig.3.2(c) can be thought of as the confirmed leader's log.

When the confirmed leader of t fails, a new term t' starts and a new leader is elected. The new leader may contain only a prefix of the speculation added by the previous confirmed leader. Fig.3.2(d) can be thought of as the log of the new leader.

The Speculation Guarantee states that if the leader of the new term t' becomes confirmed, the old speculation present in its log will become durable, and the speculation that is not present in its log will be gone permanently. Fig.3.2(e) can be thought of as the log of the confirmed leader of t': the pink portion in Fig.3.2(d) has become green (or durable). The lost speculation (the gray portion in Fig.3.2(d)) will never reappear in any future log and will be lost forever.

Further, in ED Raft, only confirmed leaders serve client requests, and the logs of all the servers mirror the confirmed leader's log.

The formal proof of the Speculation Guarantee can be found in Appendix A.3.

3.3 ED Raft in Action

We present the evolution of a three-node ED Raft cluster across two leader changes and multiple client requests in six panels. Each panel is depicted in Fig.3.3. We describe these panels below and highlight the ED Raft properties.

Each panel shows the logs for each of the three nodes. The state at each node is shown next to the logs. The logs contain some log entries. In ED Raft, log entries have a unique identifier of the form $\langle index, term \rangle$. $index$ is the position of the entry in the log of the leader that added the entry, and $term$ is the term of the leader that added the entry.

The term at each node is shown in a shaded circle next to the node's log. The first log in each panel is the logical log for a given term. A node's logical log is shaded in the same color as the node's term.

Figure 3.3: ED Raft in Action

The tag at the top-left corner of the log indicates the node state: the green check indicates the node is the leader, the green check within a green circle indicates the node is a confirmed leader, and the purple tag indicates the node is a candidate. The absence of the tag indicates the node is a follower. The red cross tag indicates a failed leader.

Panel Fig.3.3(a)

Term t is underway and node 1 is the confirmed leader for t. The leader has added four entries, of which the first three are committed, and the fourth is speculative.

The logical log for term t, depicted in green, contains all *committed*(t) entries and speculative entries added by the confirmed leader of t. Note that it looks like the log for the confirmed leader for t.

The state at each node is shown next to each log. Node 1 has applied the speculative entry to its local key-value store. This is valid as the ED Raft condition for applying speculative entry is fulfilled — the entry at the commit index has the node's current term as its term. Similarly, node 2 has applied the speculative entry in its log. Node 3 has not, although it is allowed to.

Observe that node 1 has applied all 4 entries of the logical log for t, node 2 has applied the first 3 entries, and node 3 has applied the first two entries. We see that the state at the nodes corresponds to a prefix of the logical log for t.

Panel Fig.3.3(b)

The leader for t has failed, and node 3 has become the candidate and incremented its term from t to t'. The term at node 2 has also changed from t to t'. The term at node 1 remains the same.

The logical log for the new term t', in blue, contains all *committed*(t') entries and all speculative entries added by the confirmed leader of t'. The latter is an empty set since t' does not have a confirmed leader yet. The former is the same as the set of *committed*(t) entries. Thus, the logical log for t' contains the first three entries of the logical log of t. It does not contain the fourth because the fourth entry was a speculative entry added by the confirmed leader of the previous term, not t'.

ED Raft requires that when the current term at a server changes, it must reset its local state to *initial state*, which, in our example, is an empty map. Therefore, the state at node

2 and node 3 is an empty map. An empty map means no entries are applied to the state. Therefore, the state at the two nodes reflects a prefix of the logical log for t'.

The state at node 1 remains unchanged. Since the current term at node 1 is still t, its state reflects a prefix of the logical log for the term t, depicted in green in (a).

Panel Fig.3.3(c)

Node 3 has become the leader of term t' and has added two speculative entries.

The logical log for t' remains the same. The *committed*(t') set remains the same as no new entries have been committed. The two new speculative entries are not a part of the logical log because the leader is not confirmed.

As servers can apply entries up to their commit index, nodes 2 and 3 have applied entries up to each of their commit indexes. These two entries are the first two entries of the logical log for the nodes' current term t', and so the state at the two nodes reflects a prefix of the logical log for their respective terms.

Although $\langle 3, t\rangle$ is committed and a part of the logical log for t', nodes 2 and 3 don't apply this entry because it does not satisfy the condition for applying speculative entries. On both nodes, the current term is t' but the entry at the commit index has term t, and so entries beyond the commit index cannot be applied. Similarly, node 3 does not apply the two entries it has added in t' for the same reason.

Panel Fig.3.3(d)

Node 3 has failed before it could become confirmed and term t'' has begun. The logical log for t'' is the same as the logical log for t'. Node 1 is the candidate for the term t''. The same election process as described above, repeats. Nodes 1 and nodes 2 reset their state and then follow the first rule and apply entries up to their commit index to their states.

Node 2 does not apply entry $\langle 3, t\rangle$ for the same reason as stated previously. Node 1 does because its commit pointer points to the entry. The state at node 1 and node 2 is a prefix of their respective logical logs.

Node 1's log still contains the old speculative entry from term t. In the previous panel(Fig.3.3(c)), node 1's state also contained this entry. But on term change, node 1 reset its state, wiping out the unresolved speculation from previous terms from its state.

Panel Fig.3.3(e)

Node 1 adds entry $\langle 5, t'' \rangle$ to its log and commits it by replicating it to node 2, and becomes the confirmed leader for t''. By committing this entry, it also commits the old speculative entry $\langle 4, t \rangle$.

The logical log for t'' now contains entries 4 and 5 because both are *committed*(t''), along with the first three entries.

At the time node 1 becomes confirmed, its log contained old speculation added by the leader of term t (entry 4). When node 1 becomes confirmed, it resolves all existing speculation in its log. $\langle 4, t \rangle$ is the speculation that survived the failure and will survive forever. In panel 4.1(c), if node 3 had not failed and become confirmed in term t', this old speculative entry would have been lost permanently.

Panel Fig.3.3(f)

Node 1 has added a new speculative entry and the logical log for t'' has grown accordingly. All the nodes are at term t'' and their state reflects a prefix of their respective logical logs.

Chapter 4

Implementation

We describe the implementation of ED Raft and evaluate it against a non-replicated, in-memory key-value and a Raft-backed key-value store. As ED Raft is an eventually durable variant of Raft, we base our implementation on an existing implementation of Raft. We choose etcd Raft[1] as our base Raft implementation. Based on Github stars, it is the most popular implementation of Raft and is used by systems like etcd, CockroachDB and Kubernetes.

We start by describing the etcd Raft library in §4.1, and establish that the API exposed by etcd Raft allows us to implement ED functionality as a layer on top of the library. In §4.2, we present the ED wrapper and describe our implementation of the ED layer for etcd Raft. Implementing ED as an overlay atop an existing Raft implementation gives us a simple ED Raft implementation. The modular design also allows ED semantics to be configurable.

Finally, we present the results of our evaluation in §4.3. The ED Raft-backed key-value store is at least 2x faster than the Raft-backed key-value store.

4.1 etcd Raft

etcd Raft is a library that implements the Raft algorithm. The application *proposes* new log entries to the library. Once the proposed log entry is successfully replicated in the log, the library notifies the application that the proposed log entry has been committed.

Applications using etcd Raft typically have a 3-layered architecture as depicted in

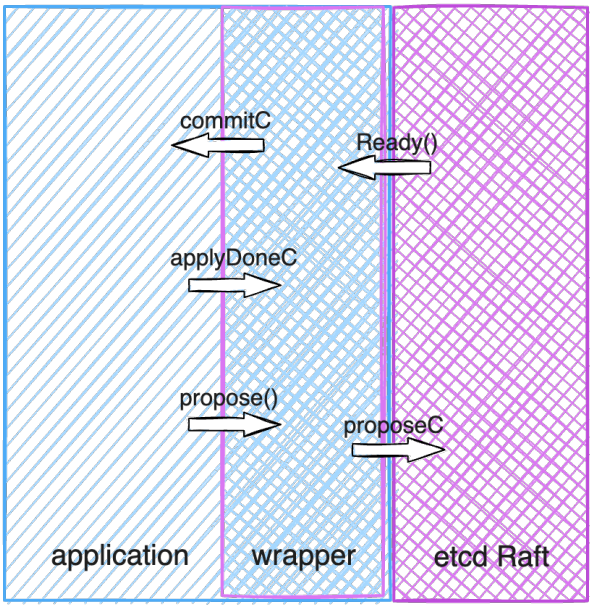

Figure 4.1: Interaction between etcd Raft, wrapper, and application

The wrapper encapsulates all interactions with the etcd Raft library and sends entries to apply to the application. The Raft wrapper sends entries when they have committed, and the ED wrapper sends speculative entries and tells the application to reset its state.

Fig.4.1. The application calls the wrapper's `propose()` function and proposes log entries, which the wrapper then forwards to the library over the `proposeC` channel.

The etcd Raft library is designed to offer great flexibility to different kinds of applications. It implements the Raft algorithmic state machine but expects the application to persistently store Raft's algorithmic state, which includes the log and variables such as the current term. This work is handled by the wrapper.

To this end, the wrapper continually calls the library's `Ready()` API. In response, the library returns a set of log entries and Raft state variables such as current term and commit index, which must be persisted. It also returns a set of log entries that are committed.

The wrapper sends committed entries in batches over the `commitC` channel. The application listens on the `commitC` channel and applies the entries it receives to the application state. Once the application has applied all entries in a given batch, it notifies the wrapper over `applyDonC`.

The wrapper is also responsible for maintaining Raft's `appliedIdx`, which tracks which log entries have been applied by the application, and updates the `appliedIdx` on receiving the done notification from the application.

4.2 ED wrapper

Our implementation of ED Raft consists of an ED wrapper on top of the etcd Raft library. In this section, we describe the implementation of the ED wrapper.

Like the non-ED wrapper described above, the ED wrapper emits entries over `commitC` and tracks `appliedIdx`. However, in addition to sending the committed entries, the ED wrapper also sends the speculative entries that can be applied according to the ED Raft protocol. Additionally, when the term changes, the ED wrapper signals the application to discard and re-initialize its state as a result of failed speculation, thereby ensuring the ED Raft safety property.

The ED wrapper can do these things because it has visibility over the speculative entries and the current term. The etcd Raft library exposes log entries and the Raft state for the wrapper to persist. The log entries are the speculative entries and the Raft state contains, among other things, the current term. Thus, ED semantics can be implemented by the ED wrapper without modifying the core etcd Raft library. We make no changes to the various interfaces: an application can have ED semantics just by swapping the wrappers.

Listing 4.1: ED Wrapper

```
while(true) {
    rd <- etcdRaft.Ready()
    // etcd Raft relies on the user to provide
    // persistent storage
    raftLog.persist(rd.Entries, rd.HardState)

    // on term change, notify application to reset state
    if (rd.HardState.Term != currentTerm) {
        resetState()
        currentTerm = rd.HardState.Term
    }

    // apply entries according to ED Raft protocol
    applyDoneC = entriesToApply(rd.committedEntries, currentTerm)

    createSnapshot(applyDoneC)
}
```

Listing 4.2: Raft Wrapper

```
while(true) {
    rd <- etcdRaft.Ready()
    // etcd Raft relies on the user to provide
    // persistent storage
    raftLog.persist(rd.Entries, rd.HardState)

    // apply committed entries
    applyDoneC = entriesToApply(rd.CommittedEntries)

    createSnapshot(applyDoneC)
}
```

Lis.4.1 presents the algorithm of the ED wrapper. In contrast, we present the non-ED wrapper (hereafter, the Raft wrapper) that exposes only committed entries for non-ED applications, in Lis.4.2. Both wrappers call `Ready()` in an event loop. The `Ready()` call returns `rd.HardState` which contains Raft state variables current term and commit index for the wrapper to persist; `rd.Entries` which is the set of log entries for the wrapper to persist; and `rd.CommittedEntries` which is the set of committed entries.

Then, the Raft wrapper calls the function to send committed entries to the application. The ED wrapper on the other hand checks if the term has changed and if it has, it notifies the application to reset its state (Lis.4.1, L9). Finally, the ED wrapper calls the function to determine which entries to send to the application.

In the next two sections, we explain the details of sending entries to the application and signaling the application to reset state on term change, and how combined these ensure the ED Raft Safety.

4.2.1 Applying speculative entries

Lis.4.3 presents the algorithm for determining which entries to send to the application. Entries are chosen according to the ED Raft protocol. These entries include all the committed entries after the applied index. Speculative entries are included only if the last entry in the committed set has the same term as the current term. Raft wrapper works the same way, except for the speculative entry generation.

The entries are sent to the application over the `commitC` channel and the applied index is set to the index of the last entry sent. The wrapper is structured such that the next batch of entries is sent only after the application is done processing the first batch. The application signals over `applyDoneC` when it is done applying all entries in a given batch. The wrapper also ensures that entries are sent to the application in the order they appear in the log and that no log entry is skipped.

Careful readers might notice that even though the library exposes speculative entries as `rd.Entries`, speculative entries to send to the application are read from the log(Lis.4.3, L7). We do this because the sets of `rd.Entries` returned by successive calls to `Ready()` contain disjoint sets of entries. If one set contains entries 1 through 5, the next will contain entries 6 through 10, and so on. Thus, every set needs to be processed and no set can be skipped. However, a set of speculative entries must be skipped if the condition for applying speculative entries is not fulfilled. There are two options to mitigate this gap issue: either buffer speculative entries until the condition is met or read speculative entries from the

Listing 4.3: Logic for sending entries to application

```
func entriesToApply(committedEntries, currentTerm) {
    // all committed entries after applied index
    ents = committedEntries[appliedIdx+1:]

    // speculative entries to send
    if ents[-1].Term == currentTerm {
        ents += raftLog[ents[-1].Index+1:]
    }

    // send entries to application
    commitC <- ents
    // update applied index
    appliedIdx = ents[-1].Index
}
```

log. We choose the latter. Additionally, we implement an in-memory log buffer that stores a portion of the log in memory as an optimization over the on-disk WAL.

4.2.2 Resetting state

When the wrapper detects that the current term has changed, it signals the application to revert to a previous durable state. This is equivalent to the ED Raft protocol. The protocol states that servers must discard their state when the current term changes. After discarding their state, according to the protocol, servers can re-apply all the committed entries.

The application reverts to a previous durable state using snapshots. A snapshot is a point-in-time copy of the application state. Since ED applications apply speculative entries, snapshots may contain speculative state. Snapshots containing speculative state cannot be used to restore state as term change may have caused speculative entries to be lost. The wrapper ensures that only snapshots containing durable state are used for restores by implementing a two-step snapshotting mechanism.

The ED wrapper asks the application to create a snapshot at regular intervals. The snapshot consists of a copy of the application state and the `snapshotIdx`. `snapshotIdx` is the index of the last log entry applied to the state before creating a snapshot. The

Listing 4.4: Reset application state on term change

```
func resetState() {
    // signal application to reload snapshot
    commitC <- nil
    appliedIdx = snapshotIdx

    // restream committed entries to the application
    commitC <- raftLog[appliedIdx+1:commitIdx]
    appliedIdx = commitIdx
}
```

ED wrapper stores this snapshot and tags it as *dirty*. As time passes and the speculative entries commit, the dirty snapshots become *stable*. The ED wrapper iterates over the list of dirty snaps at regular intervals. Snapshots having `snapshotIdx <= commitIdx` are marked stable. All the dirty snapshots are discarded when the term changes.

Lis.4.4 presents the logic for reverting the application state to a safe state using snapshots. When the term changes, the ED wrapper asks the application to load the latest stable snapshot by sending `nil` signal over `commitC`. When the snapshot is restored, the `appliedIdx` is set to `snapshotIdx`.

Further, once the application has restored the snapshot, the ED wrapper sends log entries up to the commit index to the application. This is done because the latest clean snapshot may not contain all the committed entries sent to the application before the snapshot restore. Not re-streaming these committed entries from the log may lead to the gap issue discussed in the previous section.

4.3 Evaluation

In this section, we place an eventually durable application, a key-value store, in contrast with an application that guarantees durability and another application that provides no durability guarantee. We compare their response times and present the results. The application that does not offer a durability guarantee provides the ideal case where durability-related latency is 0.

We start by describing the experiment setup and present the results of the experiment. Finally, we add the cost of to the contrast depicted in Fig.1.1.

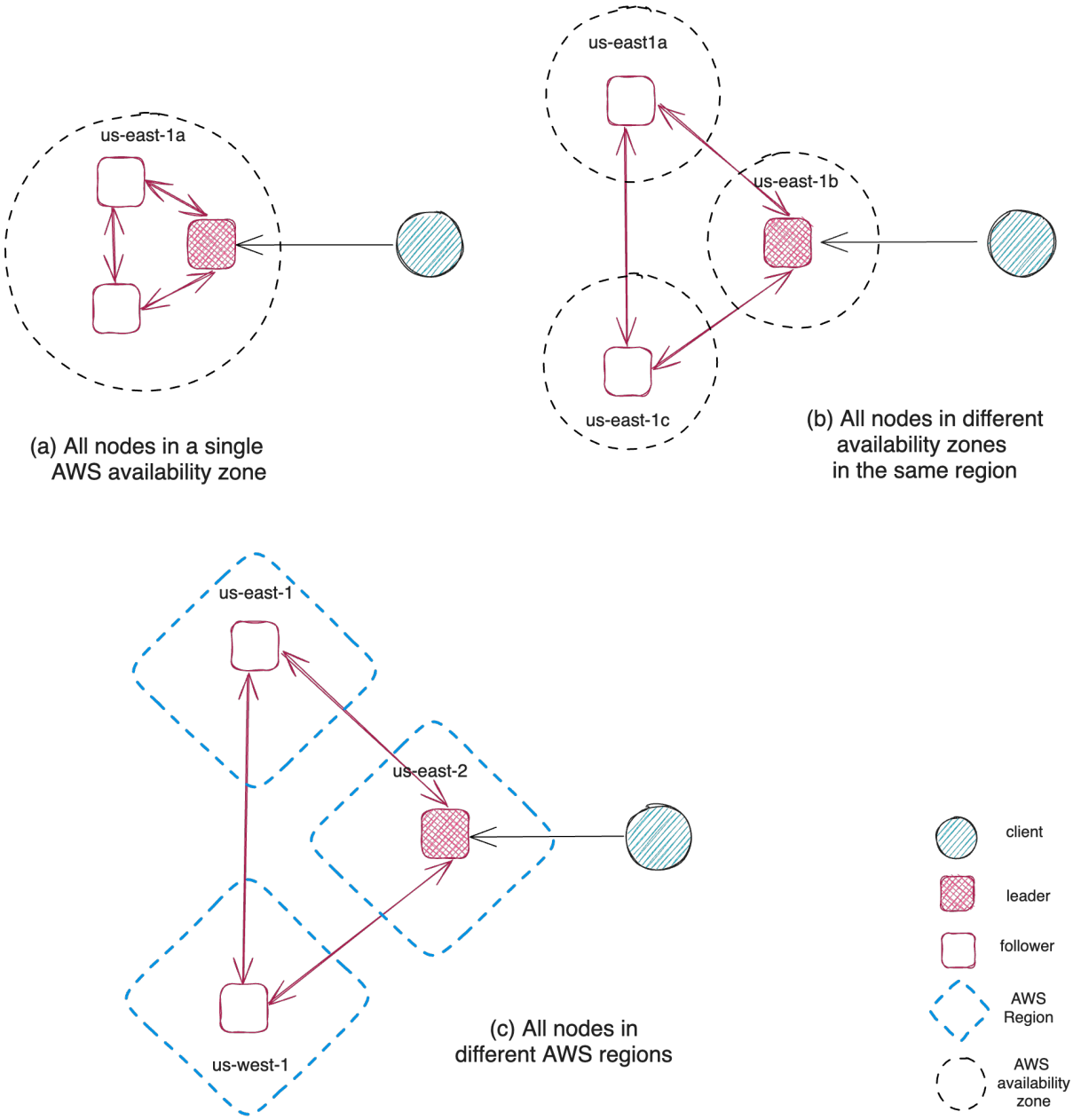

Figure 4.2: Key-value Store Node Placement

Experiment Setup

Our experiment consisted of three key-value store variants: a "no-durability" variant, an eventually durable variant, and a variant that guarantees durability. All three key-value variants are in-memory hashmaps. The no-durability variant is a single-node, in-memory hashmap. Writes to this key-value store simply insert into the hashmap and return. The eventually durable key-value store variant is backed by ED Raft, particularly the ED Raft implementation described earlier in this chapter. The key-value store variants that guarantee durability are backed by Raft, etcd Raft in particular.

Both the ED Raft and Raft-backed key-value stores consist of three nodes, with each node hosting an in-memory hashmap. When a write is performed on the Raft key-value store, it replicates the write request and performs the insert after the write commits. The ED key-value store inserts the write into the in-memory hashmap and returns, replicating the write in the background.

We ran our experiments on AWS. Each node of the key-value stores under evaluation was deployed on a separate EC2 instance of type m5.large(2 vCPU, 8 GB). For all three key-value store variants, we recorded the time it takes to sequentially insert 10,000 key-value pairs. In Raft and ED-Raft variants, inserts were performed on the leader node.

Three variations of the same experiment were done, with each variation increasing the distance between the key-value store nodes. This does not apply to the no-durability variant as it is a single-node key-value store. Figure 4.2 depicts the different placements of the nodes in the Raft- and ED Raft-backed key-value stores. The placement of the client depicts that inserts were performed on the leader node. In the first variation of the experiment, the nodes were placed closest together, in the same AWS availability zone(a). The second variation placed the nodes in different availability zones(b) and the third-placed nodes farthest apart, in different AWS regions(c). Three iterations of each experiment were done, with every iteration performing 10,000 write requests. The response time was recorded for each request and averaged over 10,000 requests.

Results

Figure 4.3 depicts the average time taken to perform an insert for each of the three key-value stores in each of the three placement configurations. As expected, the ED key-value store is faster than its strongly durable counterpart. It is at least twice as fast when the nodes are closest together, i.e., in the same AZ.

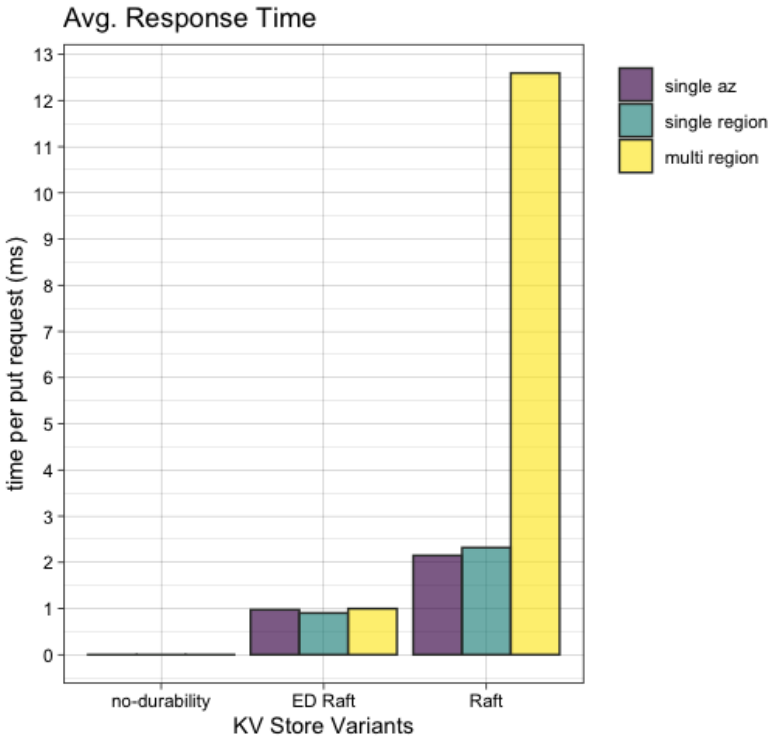

Figure 4.3: Response time for different key-value store configurations

The ED Raft key-value store is at least twice as fast as the key-value store that guarantees durability. The difference in the performance becomes significant as the distance between the key-value store nodes increases.

As the distance between the key-value store nodes increases, the response time for the ED key-value store remains unchanged but the response time for the durable key-value store increases. This is expected because ED Raft does not wait for entries to commit before applying them, essentially eliminating critical path network calls that are unavoidable in Raft. The performance gap between the ED Raft key-value store and the durable key-value store becomes significant as the cluster widens.

The ED key-value store is slower than its no-durability counterpart: an insert into the ED key-value store takes 1 ms compared to the no-durability variant which is practically instantaneous. This 1 ms is the overhead of ED Raft machinery. The no-durability variant simply inserts into the in-memory hashmap and returns. On the other hand, the ED key-value store triggers ED Raft to start the replication but inserts and returns without waiting for the replication to complete.

Eventual Durability Savings

In Fig.1.1, we presented the cost of durability by comparing the response times in a key-value store that does not make writes durable (no-durability) and three key-value stores that guarantee durability. We now add response time for an eventually durable key-value store to this contrast. Fig.4.4 represents the cost saved by eventual durability. This is just a different presentation of the results from the experiment described previously.

The three key-value stores that guarantee durability differ in their node placement. The variant having nodes placed in the same AZ represents "small-strength durability" (Fig.4.2a), the variant having nodes placed in different AZ in the same region represents "medium-strength durability" (Fig.4.2b), and the variant having nodes placed in different AWS regions represents "high-strength durability" (Fig.4.2c).

Eventual durability helps shave at least 1 ms from the response time. Typically, applications are deployed in medium or high durability configurations, in which case the saving due to eventual durability is more than half.

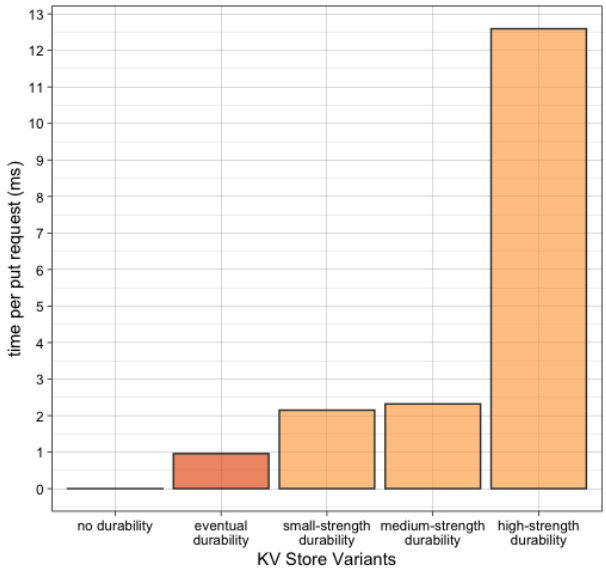

Figure 4.4: Eventual Durability Savings

Eventual durability helps shave at least 1 ms from the response time.

Chapter 5

Related Work

Durability Guarantees

While it is commonly acknowledged that durability is expensive, the durability aspect of systems remains largely unexplored. Some of the only works that discuss examine the durability aspect are WheelFS[26], CAD[12], Eventual Durability or ED[15], and CAP-for-durability[2]. WheelFS is a distributed filesystem that allows clients to specify durability "strength" for a file. CAD offers high performance by delaying durability and shifts the durability point from writes to reads: data is made durable before it is read. CAP-for-durability is a blog post that examines system behaviors across two planes: the durability–latency plane and the durability–availability plane. These works don't discuss data loss beyond acknowledging that their design is vulnerable to data loss.

Similar to the Eventually Durable State Machine (EDSM), the ED model[15] allows applications to make performance/durability trade-offs while managing the blast radius of data loss caused by failures. The ED model allows reasoning about data loss in the context of transactions while EDSM allows reasoning about data loss in replicated systems.

Operational Data Loss

Outages are common in operational environments, often resulting in data loss[4]. Data loss is tolerated as an unfortunate inevitability to the extent that service level agreements (SLAs) incorporate metrics like the Recovery Point Objective (RPO) to assess such losses. There may be a variety of causes for this data loss including storage failures.

While a variety of work caters to mitigating storage failures[10, 8, 6, 5], to the best of our knowledge, only our work and the ED work establish a model to reason about the data loss itself. We make this statement while recognizing the specific context of data loss in EDSM.

Unix Filesystem

The Unix write API is similar to ours in the sense that writes may not be immediately durable. A successful `write` does not automatically guarantee that data is persisted on disk[3]. `fsync` or `close` must be performed to guarantee persistence. A crash may lead to data loss and `fschk` must be performed after a crash to get information about the lost files.

Efficient Replication

Single-node systems eliminate the replication overhead but they don't scale. Using specialized network hardware such as RDMA[29, 28] is one way to speed up the replication process.

Approaches optimizing replication on commodity hardware typically focus on reducing the number of network round-trips required for coordination, either by algorithms that allow batching multiple requests[27] or by predicting the ordering of requests[32].

While Raft achieves consensus in a single round-trip, FlexiRaft[31] points out performance bottlenecks in Raft and proposes configurable quorums to allow for a range of latency, throughput, and fault tolerance trade-offs. Another way to reduce replication-induced latency is to optimize consensus: by using an optimized quorum strategy[11] or by offloading partial consensus functionality to the network for performance[33]. Other approaches avoid wide area coordination by co-locating clients and data[24, 9] or by leveraging partial order[17].

All approaches described above focus on optimally achieving the durability goal and don't stray into the weak durability domain. These approaches are orthogonal and complementary to the EDSM.

Speculative Execution

Given the temporal spread of the works in this domain[18, 14, 16, 27, 23, 30, 19], the idea of executing requests speculatively before a final order has been established has been

around for a while. If the final order is different than the predicted order, the command is rolled back and redone. Speculative execution systems have two flavors: those that expose clients to the speculative state[30, 19] and those that don't[18, 14, 16, 27, 23].

Systems that don't expose the client to the result of the speculative execution rely on the insight that in the vast majority of cases, the predicted order will be the same as the final order. Therefore, the execution of the command and the execution of the protocol can overlap in time, thereby reducing latency for the client.

SpecBFT[30] and SpecFS[19] expose clients to the speculative state, like the EDSM. SpecBFT, SpecFS and EDSM share the common insight that in most cases speculation will commit and low latency responses can be achieved by not waiting for confirmation.

The main risk with exposing speculative state is that it may change, requiring the client to track and revert any actions dependent on the speculative state. SpecBFT and SpecFS achieve this by having the server and client match dependency contexts. Whenever the client makes a request, it includes the context of earlier speculative operations the request depends on. The server executes a request if the chain of dependency of the request has not changed. The client also tracks the status of each speculative operation it performs: if an operation fails, the client rolls it back along with all dependent operations. EDSM on the other hand, provides the *sync* operation as a tool for the client to manage the risk associated with exposing speculative state.

Speculative execution that externalizes the speculative state is inherently indeterministic. SpecBFT and SpecFS provide a model for the client to continue executing around the non-determinism. SpecBFT and SpecFS expose enough system state for the client to make modifications as a result of failed speculation. EDSM does not. It only provides tooling to control the risk of speculation. EDSM is primarily an approach to weaken the durability guarantee and provides a model to reason about the inherent risk of data loss. However, exposing failure information like SpecBFT and SpecFS is useful and context matching is an effective solution that the EDSM can adopt as well.

Eventual Guaranteed Execution

Systems in this category provide fast response by eliminating the latency introduced due to request ordering. These systems guarantee request durability. Acknowledgment of a request represents a guarantee that the request will be executed but the acknowledgment does not contain execution results. Nilext[13] guarantees that the request will be executed but defers its ordering and execution, thereby reducing latency by cutting out time

spent in coordination. Similarly, Weave[11] decouples request execution from the execution guarantee and also associates a stable sequence number to each request.

The insight behind these approaches is that in a geo-replicated system, total order requires at least one wide area network call. By deferring ordering, this time is saved. In ED Raft, the backbone of our EDSM implementation, the leader acts as a sequencer and so replication is achieved in a single network round trip.

Chapter 6

Conclusion

The Eventually Durable State Machine is a general approach to allow applications to make durability/latency tradeoffs and provides a model to reason about the inevitable data loss. ED applications are fast with weak durability guarantees. Writes to an ED application are not durable when they return. Writes may become durable eventually, sometime after they return.

We define the ED Log, an abstract entity that represents the client requests applied by the ED state machine at any given time. The behavior of the ED Log represents ED semantics. We develop ED Raft to support the ED Log behavior and thus implement the ED state machine.

ED Raft is a log-based protocol that implements the ED state machine. ED Raft is the eventually durable variant of the Raft consensus algorithm[22]. ED Raft does not wait for client proposals to replicate before acknowledging them, as opposed to Raft which guarantees durability by responding to client requests only after they have been replicated.

We implement ED Raft as a wrapper on top of etcd Raft, a popular, widely-used Raft implementation. Our design choices in implementing ED Raft allow any application using etcd Raft to seamlessly adopt the ED model.

We evaluate an eventually durable key-value store (backed by ED Raft) against a non-replicated and a key-value store that guarantees durability (backed by Raft). The ED key-value store is *at least* 2x faster than the key-value store that guarantees durability.

Future Work

Replication lies at the heart of durability. Raft is one way to achieve replication. A variety of protocols exist to achieve replication and consensus in a distributed setup, each optimizing for a different set of parameters. Paxos and its various flavors, and Zookeeper are particularly common. ED variants of these common replication techniques would not only make it easier for the applications using them to utilize the ED model but would also be a good exercise in understanding the landscape of challenges in replication and extending the ED model to account for them.

While the ED model allows an application to be fast, it leaves the application vulnerable to data loss. The application may accept the risk of data loss but data loss remains undesirable. And so application may implement mechanisms to counter the data loss, such as maintaining a buffer of replication-pending requests. Implementation of these counter mechanisms is non-trivial as these counters would need to reconcile the buffer with the storage state after failures. Additionally, these counters would need to be implemented such that they don't take away from the benefits provided by ED, i.e., fast user response.

Further, a generic data loss countering layer that applications can import as a module would be beneficial. It would abstract away the logistics of data loss management from the application, thus simplifying the adoption of the ED model. The generic solution would need to consider the spectrum of state representations applications can have.

References

[1] etcd raft implementation. `https://github.com/etcd-io/raft`. Accessed: 2024-02-26.

[2] Is there a cap theorem for durability? `https://brooker.co.za/blog/2015/09/26/cap-durability.html`. Accessed: 2024-04-09.

[3] Linux manual page for `write`. `https://man7.org/linux/man-pages/man2/write.2.html#notes`. Accessed: 2024-05-08.

[4] Rpo and rto: getting to zero downtime and zero data loss. `https://www.cockroachlabs.com/blog/demand-zero-rpo/`. Accessed: 2024-04-09.

[5] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Protocol-Aware recovery for Consensus-Based storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 15–32, Oakland, CA, February 2018. USENIX Association.

[6] William J Bolosky, Dexter Bradshaw, Randolph B Haagens, Norbert P Kusters, and Peng Li. Paxos replicated state machines as the basis of a {High-Performance} data store. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[7] Sebastian Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1–2):1–150, oct 2014.

[8] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

[9] Paulo Coelho and Fernando Pedone. Geopaxos+: Practical geographical state machine replication. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 233–243, 2021.

[10] Miguel Correia, Daniel Gómez Ferro, Flavio P Junqueira, and Marco Serafini. Practical hardening of {Crash-Tolerant} systems. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 453–466, 2012.

[11] Michael Eischer, Benedikt Straßner, and Tobias Distler. Low-latency geo-replicated state machines with guaranteed writes. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Strong and efficient consistency with Consistency-Aware durability. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 323–337, Santa Clara, CA, February 2020. USENIX Association.

[13] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting nil-externality for fast replicated storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 440–456, New York, NY, USA, 2021. Association for Computing Machinery.

[14] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 477–484, 2002.

[15] Tejasvi Kashi. Eventual durability of ACID transactions in database systems, August 2023. Master's Thesis, UWSpace, `http://hdl.handle.net/10012/19705`.

[16] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, pages 424–431, 1999.

[17] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The FuzzyLog: A partially ordered shared log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 357–372, Carlsbad, CA, October 2018. USENIX Association.

[18] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. High performance state-machine replication. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 454–465, 2011.

[19] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. *SIGOPS Oper. Syst. Rev.*, 39(5):191–205, oct 2005.

[20] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 191–205, New York, NY, USA, 2005. Association for Computing Machinery.

[21] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, Department of Computer Science, 2014.

[22] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[23] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 43–57, Oakland, CA, May 2015. USENIX Association.

[24] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *2010 29th IEEE Symposium on Reliable Distributed Systems*, pages 214–224, 2010.

[25] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990.

[26] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, M. Frans Kaashoek, and Robert Morris. Flexible, Wide-Area storage for distributed systems with WheelFS. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*, Boston, MA, April 2009. USENIX Association.

[27] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings*

of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. Tailwind: Fast and atomic RDMA-based replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 851–863, Boston, MA, July 2018. USENIX Association.

[29] Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu. Replicating persistent memory Key-Value stores with efficient RDMA abstraction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 441–459, Boston, MA, July 2023. USENIX Association.

[30] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, page 245–260, USA, 2009. USENIX Association.

[31] Ritwik Yadav and Anirban Rahut. Flexiraft: Flexible quorums with raft. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023.

[32] Xinan Yan, Linguan Yang, and Bernard Wong. Domino: using network measurements to reduce state machine replication latency in wans. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 351–363, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Yang Zhang, Bo Han, Zhi-Li Zhang, and Vijay Gopalakrishnan. Network-assisted raft consensus algorithm. In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, page 94–96, New York, NY, USA, 2017. Association for Computing Machinery.

APPENDICES

Appendix A

ED Raft Safety Proof

This appendix includes proofs of the ED Raft Safety and the Speculation Guarantee discussed in §3.

A.1 Conventions

The ED Raft proof borrows the following conventions used in the original dissertation that presented Raft (Appendix B.1, [21]).

- foo' has a specific meaning: the value of variable foo in the next state of the system.

A.2 Definitions & Terminology

The ED Raft proof makes use of the following definitions and results from the original dissertation that presented Raft (Appendix B.3, [21]). These results also hold for ED Raft because ED Raft differs from Raft in only the way log entries are applied to the state machine. Leader election and replication in ED Raft remain the same as in Raft.

Definition R1 (Definition 1, [21]). An entry $\langle index, term \rangle$ is **committed at term** t or $committed(t)$ if it is present in every leader's log following t.

Definition R2 (Definition 2, [21]). An entry $\langle index, term \rangle$ is **immediately committed** if it is acknowledged by a quorum (including the leader) during $term$.

Definition R3 (Definition 3, [21]). An entry $\langle index, term \rangle$ is **prefix committed at term** t if there is another entry that is $committed(t)$ following it in some log.

Lemma R4 (Lemma 4, [21]). An $\langle index, term \rangle$ pair identifies a log prefix.

Lemma R6 (Lemma 6, [21]). A server's current term is always at least as large as the terms in its log.

Lemma R7 (Lemma 7, [21]). The terms of entries grow monotonically in each log.

Lemma R8 (Lemma 8, [21]). *Immediately committed* entries are *committed*.

Lemma R9 (Lemma 9, [21]). $Prefix$ *committed* entries are *committed* in the same term.

The following are definitions specific to ED Raft.

Definition 4. A log entry $\langle index, term \rangle$ that is not immediately committed is **speculative**.

$$
\begin{aligned}
speculative \triangleq \{ & \langle index, term \rangle : \\
& \exists\ log \in allLogs : \\
& \quad \langle index, term \rangle \in log \\
& \quad \wedge \langle index, term \rangle \notin immediatelyCommitted \ \}
\end{aligned}
$$

Definition 5. A term has a **confirmed leader** if it has at least one immediately committed entry.

$$confirmedLeaderTerms \triangleq \{t : \\ \exists \ \langle i, t\rangle \in immediatelyCommitted \ \}$$

Definition 6. The **logical log for term *t*** contains all *committed*(t) entries and all speculative entries added by the confirmed leader of term t. Let $C, C \geq 0$ be the number of entries in *committed*(t), and $S, S \geq 0$ be the number of speculative entries. Then, the logical log for t has C entries in positions $[1...C]$ and the S speculative entries in positions $[C + 1...C + S]$.

$$logicalLog(t) \triangleq \{committed(t) \\ \vee \langle i, t\rangle : \\ t \in confirmedLeaderTerms \ \}$$

A.3 Speculation Guarantee

Theorem 1. Speculation Guarantee. Let $\langle i, t\rangle$ be a speculative log entry. Then, either

1. $\forall\ t' > t$: if t' has a confirmed leader, then $\langle i, t\rangle \in$ *committed*(t'), or
2. $\forall\ t' > t$: if t' has a confirmed leader, then $\langle i, t\rangle$ is not present in that confirmed leader's log.

Sketch. If the speculative entry is present in the log of the confirmed leader, it will be committed. If not, then, because terms increase monotonically, the entry will not appear in the said confirmed leader's log, and thus, in future leaders' logs.

Proof. We consider two cases:

1. Case: Let $t^* > t$ be the first term for which there is a confirmed leader, and suppose that the speculative entry $x = \langle i, t\rangle$ is present in the log of term t^*'s confirmed leader.
   (a) Term t^* has a confirmed leader. Therefore, by definition, there exists an immediately committed entry $y = \langle j, t^*\rangle$ in the log. By Lemma R8, $y \in$ *committed*(t^*).
   (b) By Lemma R7, since the terms of entries in a log grow monotonically, entry x is present before entry y in the log.

(c) By the definition of prefix committed and Lemma R9, $x \in committed(t^*)$. Thus, by definition of committed, $x \in committed(t'), \forall t' > t^*$.

2. Case: Let $t^* > t$ be the first term for which there is a confirmed leader, and suppose that the speculative entry $x = \langle i, t \rangle$ is not present in the log of term t^*'s confirmed leader.

   (a) Term t^* has a confirmed leader. Therefore, by definition, there exists an immediately committed entry $y = \langle j, t^* \rangle$ in the log. By Lemma R8, $\langle j, t^* \rangle \in committed(t^*)$.

   (b) Case: $i < j$

      i. x is not in the log. Thus, another entry $z = \langle i, t^o \rangle$ is present at index i.

      ii. Since $y \in committed(t^*)$, it will be present in the leader's log $\forall\ t' > t^*$.

      iii. By Lemma R4, y uniquely identifies the log prefix which includes z at index i.

      iv. Since y is present in the logs of leaders $\forall\ t' > t^*$, z will also be present. As z is present, x will not be present any of these logs.

   (c) Case: $i > j$

      i. Since $y \in committed(t^*)$, it will be present in the leader's logs $\forall\ t' > t^*$.

      ii. By Lemma R7, the terms of entries grow monotonically in the leader's log. Since $x.term < t'$, x will not be present in the leader's log $\forall\ t' > t^*$.

□

A.4 State Machine Safety

Lemma 1. The logical log for term t grows monotonically.

$$\forall\ e \in elections :$$
$$\forall\ index \in 1...Len(logicalLog(e.term)) :$$
$$logicalLog'[e.term][index] = logicalLog[e.term][index]$$

Sketch. C never shrinks. S shrinks only when entries move from S to C.

Proof. As the logical log is $C + S$, the size of the logical log can only shrink if C shrinks or S shrinks. We consider each case:

1. Case: C shrinks. This does not happen because entries never leave $committed(t)$.

2. Case: S shrinks. This can only happen if a speculative entry having term t becomes immediately committed and joins $committed(t)$, by Lemma R8. Thus, an entry leaving S joins C and $C + S$ remains the same.

□

Theorem 2. State Machine Safety. The state of a server reflects a prefix of the logical log for the server's term.

$$\forall\, i \in Server :$$
$$\quad \forall\, index \in 1...appliedIndex[i] :$$
$$\quad\quad log[i][index] = logicalLog[currentTerm[i]][index]$$

Sketch. Servers apply entries up to their $commitIndex$ which covers all or a subset of $committed(currentTerm)$ entries. Servers apply entries after their $commitIndex$ if their $currentTerm$ has a confirmed leader. At term change, servers set their state to $initialState$ and their $appliedIndex$ to 0.

Proof by induction on an execution.

1. Base case: Invariant trivially holds as state is *initial state* and $appliedIndex[i] = 0$.

2. Case: state at server i changes and $appliedIndex[i]$ is incremented.

   (a) Let $currentTerm[i] = t$. The logical log for t contains all $committed(t)$ and speculative entries added by the confirmed leader of term t.

   (b) Case: $appliedIndex[i] \leq commitIndex[i]$.

      i. Entry at $commitIndex[i] \in committed(t)$. By Lemma R9, all entries up to $commitIndex[i] \in committed(t)$.

   (c) Case: $appliedIndex[i] > commitIndex[i]$.

      i. Entry at $commitIndex[i]$ has term t, and by definition, is an immediately committed entry. Therefore, t has a confirmed leader.

      ii. By Lemma R6 and R7, entries after $commitIndex[i]$ will only have term t. Either these entries are $committed(t)$ and $commitIndex[i]$ is not updated to reflect that, or these are speculative entries added by the confirmed leader of term t.

(d) $appliedIndex[i]$ will be decremented only when $currentTerm[i]$ changes. This is handled below.

3. Case: the $currentTerm$ at server i changes.

   (a) The protocol sets a server's state to *initial state* and $appliedIndex[i] = 0$. Invariant trivially holds.

4. Case: the logical log for a given term t changes.

   (a) According to Lemma 1, t's logical log will only grow, and $logicalLog(t)$ will be a prefix of $logicalLog'(t)$.

   (b) Since invariant holds for $logicalLog(t)$, invariant will hold for $logicalLog'(t)$.

□