FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Computer Science	Spring semester, 2017 Open / ~~Confidential~~
Author: Sebastian Mæland Pedersen	Sebastian M. Pedersen …………………………………… (signature author)
Instructor: Hein Meling Supervisor(s): Hein Meling, Leander Jehl and Tormod Erevik Lea	
Title of Master's Thesis: A Practical Analysis of the Gorums Framework A Case Study on Replicated Services with Raft	
ECTS: 30	
Subject headings: Distributed Systems • Reconfiguration • Consensus • State Machine Replication • Gorums • Raft • Paxos	Pages: 76 + attachments: Code included in PDF Stavanger, June 15, 2017

A Practical Analysis of the Gorums Framework

A Case Study on Replicated Services with Raft

Sebastian M. Pedersen

June 2017

University of Stavanger

Department of Electrical Engineering and Computer Science
Faculty of Science of Technology
University of Stavanger

Abstract

Gorums is a novel RPC framework developed to make it easier to build fault tolerant distributed systems. We want to assess whether Gorums can simplify the implementation of a practical fault tolerant service that supports reconfiguration. The Raft consensus algorithm is implemented in Gorums with the ability to do single-server configuration changes. In addition, we perform a background study of two state of the art Raft implementations. The abstractions used in these implementations are then compared to the abstractions Gorums provides and how they are used in our Raft implementation. A service is created that can be used with any of the aforementioned Raft implementations for consistency and fault tolerance. This service is then used to evaluate the different implementations through experimentation. Our evaluation shows that the Raft implementation that uses Gorums perform better with regards to latency and overall throughput during normal operation.

We do however discover this implementation to be sensitive to omission faults, which can further lead to availability issues if not handled properly. We solve this by developing extensions to Raft and Gorums. We show that these methods perform on a similar level when compared with the state of the art implementations. Results from our implementation efforts indicate that Raft's log replication process is problematic to implement with Gorums' abstractions. We discover that this is due to Raft adopting a monolithic design aimed to reduce the number of different RPC types, breaching the separation of concerns design principle.

Acknowledgments

I would like to thank my supervisor, Professor Hein Meling, for his guidance and invaluable feedback throughout the course of this thesis.

I would also like to express my gratitude toward Leander Jehl for always asking the right questions and forcing me to think outside of the box.

Finally, I would like to thank Tormod Erevik Lea for helping me understand and make further contributions to the Gorums framework.

Contents

Abstract **ii**

Acknowledgments **iii**

1 Introduction **1**
1.1 Contributions and Outline 2

2 Background **4**
2.1 State Machine Replication 4
2.2 Consensus 5
2.3 Reconfiguration 5
2.4 The Paxos Protocol 6
2.5 The Raft Algorithm 7
2.5.1 Leader Election 8
2.5.2 Log Replication 9
2.5.3 Reconfiguration 11
2.5.4 Batching and Pipelining 13

3 Gorums **14**
3.1 Configurations and Quorum Calls 14
3.2 Quorum Functions 15
3.3 Modifiers 17
3.3.1 Custom Response Type 17
3.3.2 Per Server Map Function 18
3.3.3 Correctables 18
3.4 Implementation Details 19
3.4.1 Cancellation 20

4 Survey of Raft Implementations **21**
4.1 Raft Implementations 21
4.1.1 etcd/raft 22

4.1.2 hashicorp/raft 22
4.2 Reconfiguration 22
4.2.1 hashicorp/raft 23
4.2.2 etcd/raft 23
4.3 Communication Abstractions 24

5 Design and Methodology **25**
5.1 Evaluation Basis for Gorums 25
5.2 System Architecture 26

6 Implementing Raft with Gorums **28**
6.1 Changes to Raft 28
6.2 Leader Election 29
6.2.1 RequestVote RPC 29
6.2.2 RequestVote Quorum Function 31
6.3 Log Replication 32
6.3.1 AppendEntries RPC 33
6.3.2 AppendEntries Quorum Function 34
6.3.3 Raft Catchup Extension 35
6.4 Reconfiguration 37

7 Implementation Challenges **39**
7.1 Canceled Replies 39
7.2 Ignored Replies 40
7.3 Reordered Requests 43
7.4 Failure Detection 46
7.5 Retaining Leadership 46
7.6 Correctables 47

8 Experimental Evaluation **49**
8.1 Code Coverage 49
8.2 Experimental Setup 50
8.3 Common Case Operation 51
8.4 Network Partition 53
8.5 Reconfiguration Policy 56
8.5.1 gorums/raft 58
8.5.2 hashicorp/raft 58
8.5.3 etcd/raft 60

9 Conclusion and Further Work **61**
9.1 Conclusion 61
9.2 Further Work 62

A Experimental Data **64**

B Attachments **66**

Bibliography **66**

1 Introduction

Understanding, designing and building distributed systems that are both fault tolerant and strongly consistent is difficult. To achieve the high availability expected from modern server applications, it is often desirable to be able to replace failed servers without disrupting the user's experience. This further complicates the development and maintenance of these systems. Thus it becomes important to provide developers with the appropriate aids to ensure correct implementation and operation.

We look at two works which strive to alleviate some of these difficulties. First the Raft consensus algorithm [32], which was developed as an easier to understand alternative to Paxos [20]. Paxos has long been the industry standard for solving consensus. It has however grown a reputation of being notoriously difficult to understand and implement correctly. The reason for this is that the original Paxos descriptions [20, 21] leave out a lot of implementation details, in the interest of making it easier to understand and prove its correctness. Next we use Gorums [24], a novel *remote procedure call* (RPC) [5] framework which provides several useful abstractions for building fault tolerant quorum-based systems.

The overall aim for this thesis is to put Gorums to the test, and assess whether the abstractions that it provides are useful in the design and implementation of distributed systems. To this end, we develop an implementation of Raft with Gorums as the backing algorithm for a fault tolerant and consistent key-value store.

This thesis builds on an initial attempt to implement Raft with Gorums as part of a capstone project. A short paper [36] was written that touches

on how log replication in Raft with Gorums is not optimal. We later discovered that the approach taken quickly leads to serious availability issues. Log replication is driven by the AppendEntries RPC, at the core of the Raft algorithm. Thus, much of the prospect for Gorums to be used with Raft lies in the following question:

- Can the abstraction in Gorums be used with the Raft AppendEntries RPC without causing serious availability issues and while maintaining acceptable performance?

As we wanted to create a system that can work in practice, we put in much effort to answer and develop an appropriate solution to this question. In Raft, reconfiguration is tightly coupled with the log replication process. Hence, solving log replication gives us the foundation that we need to implement reconfiguration. Reconfiguration is implemented and evaluated in the form of single-server membership changes.

In addition to our own Raft implementation we study two state of the art Raft implementations. Furthermore, we develop support for these implementations as alternative backing *replicated state machines* (RSMs) [40] to our fault tolerant service. As a result, we can run experiments that compares the implementations from the same frame of reference. We use this to evaluate the solutions we develop in the creation of Raft with Gorums.

1.1 Contributions and Outline

In summary, in this thesis we make the following contributions:

- We create a replicated service that through a common *application programming interface* (API) can operate with different replicated state machine implementations. We provide three such implementations, all using the Raft consensus algorithm.
- We fully develop one of these implementations, including support for reconfiguration, with Gorums. We use this implementation effort to show the usefulness of the abstractions Gorums provides, and whether they help with the implementation of reconfiguration.
- We complete a survey of the code for two state of the art Raft implementations, identifying commonly developed abstractions, seeing how reconfiguration is implemented and comparing our findings to the abstractions Gorums provides.

- We perform a thorough analysis of Raft with Gorums, presenting the challenges experienced and the solution developed.
- We create the *Catchup* extension to the Raft algorithm and the *Strict Ordering* option for Gorums, demonstrating that log replication in Raft can be implemented without availability problems using Gorums.
- Common case operation, the Catchup extension, and the Strict Ordering option is validated against the state of the art implementations through experimental evaluation.
- We make numerous contributions to the Gorums framework.

The remainder of this thesis is structured as follows:

Chapter 2 introduces relevant background material with a focus on Raft and how it relates to Paxos.

Chapter 3 provides an introduction to the Gorums framework, its abstractions and a few of the available options used and discussed in this thesis.

Chapter 4 presents a code survey for two state of the art Raft implementations. Identifying commonly developed abstractions and how reconfiguration is implemented.

Chapter 5 provides an overview of our methodology and the overall system architecture.

Chapter 6 describes how the Raft consensus algorithm is implemented with Gorums.

Chapter 7 lists challenges encountered during the implementation of Raft with Gorums and the solutions that we develop.

Chapter 8 provides an experimental evaluation of the Raft implementations and our proposed solutions for problems encountered in Raft with Gorums.

Chapter 9 concludes and presents suggestions for further work.

2
Background

This chapter presents background material related to this thesis. We start by presenting the core concepts: state machine replication, consensus and reconfiguration. Further we give an introduction to the Paxos protocol and the Raft algorithm.

2.1 State Machine Replication

A state machine is an abstract machine that can be in exactly one state at a time. The machines state can be modified through external inputs. Given an input, the state machine transitions into a new state based on its current state and the input provided. This process is deterministic. To clarify, given a state machine with a specific state, the same sequence of inputs will always provide the same sequence of state transitions.

When creating a fault tolerant service, the state machine approach [40] is often used. The idea is to use several state machines that are fed the same sequence of inputs. Hence, if some state machines fail, any of the other available machines can be used. The difficulty of this approach comes from the fact that we need a reliable way to deliver the exact same stream of inputs to each machine. Typically this is achieved using a consensus protocol.

2.2 Consensus

Consensus is a fundamental problem of distributed computing. Given a set of servers proposing values, it is the problem of deciding on one value among the servers. Due to the asynchronous nature of a distributed system, servers may crash or become unavailable for longer periods of time, some safety criteria are required [21]:

- Only a value that has been proposed can be decided on.
- Only a single value can be decided on.
- A server never learns that a value has been decided on unless it actually has been.

This ensures that all servers decide correctly, if they ever decide. In a practical system we would also like the ability to actually make progress. In an eventually synchronous system, progress can be made given that a majority of the servers are functioning [22]. We will often refer to a majority as a *quorum* or the quorum size of the system.

2.3 Reconfiguration

Reconfiguration, or membership change, is the act of modifying the set of servers making up a distributed system. This is useful if you need to replace faulty servers or change the replication factor. A new configuration can be applied manually by taking all the servers offline, and starting only the servers making up the new configuration. However, this causes the system to be unavailable while the reconfiguration is taking place, that is often not desirable. There is also the possibility of human error when changing the configuration manually.

A preferable approach to reconfiguration is to automate it. In practice it would be beneficial to do the change at runtime, while servicing client requests, to preserve availability. Safely adding or removing servers from a running consensus system is a complex process. As different servers discover the new configuration at different times, there will be periods where inconsistencies can occur unless care is taken. We will expand on this in Section 2.5.3, where we introduce reconfiguration in Raft.

2.4 The Paxos Protocol

Paxos is a consensus algorithm satisfying the safety criteria specified in Section 2.2. The original description [20] received a lot of critique for being too difficult to understand, which led the author to publish a paper with a simplified explanation [21].

This paper describes the algorithm through the interaction of three classes of agents: *proposers*, *acceptors* and *learners*. These agents can be individual servers but usually a single server contains the logic for all three agents.

Proposers are responsible for proposing a value that all the servers should agree on. Next, acceptors accept a value among those proposed, and if enough accepts agree on the same value, that value is said to be chosen. Finally, learners learn values that are chosen.

The algorithm operates in two phases. The first phase is used to determine if a value has already been chosen. If no value has been chosen we can propose any value. Otherwise we obtain the chosen value and propose that. It works as follows:

- Each proposer have the ability to assign a increasing unique *round* number to each proposal. This is often done by assigning each proposer an unique identifier $i \in \mathbb{N}$. New round numbers are then generated with $p = i + x \cdot n$, where n is the number of proposers, and x is the total number of proposals made by that proposer.

- A proposer selects its next round number and broadcasts it to the acceptors, in a *prepare* message.

- If an acceptor receives a prepare message with a round number greater than any round number it has seen so far, it responds with a *promise* not to accept any lower round numbers. If the acceptor have accepted a value in the past, the promise must contain the previous round number and value.

- If the proposer receive a majority of promises containing a value, it must propose the value from the highest round number contained among the received promises. If no prior values are received, the proposer is free to propose its own value.

The next phase tries to get the value accepted:

- The proposer broadcasts an *accept* messages containing the value picked in the first phase and the assigned round number.

- If an acceptor receives an accept message, it accepts the proposed value unless it has seen a higher round number.
- When an acceptor accepts a value, it broadcasts a *learn* message with the round number and value to every learner.
- On receiving a learn with the same round number from a majority of the acceptors, a learner learns that the value received is chosen.

To create an ordered log, as mentioned in Section 2.1, we can implement a sequence of separate executions of Paxos. Practical systems typically implement an optimization. A leader is chosen to function as the sole proposer. If the leader fails, a new leader is chosen. Thus, the first phase of Paxos is only required during leader changes. This optimization is called multi-Paxos.

2.5 The Raft Algorithm

Raft [32, 33] is a consensus algorithm developed as an easier to understand alternative to Paxos. Raft is a leader-based algorithm similar to ViewStamped Replication (VR) [26, 28] and ZooKeeper Atomic Broadcast (Zab) [19]. Raft is explicit about managing a replicated log, i.e., it produces an equivalent result to multi-Paxos. There exists a multitude of Raft implementations [31] in various stages of development and quality, and there have been an attempt [17] to repeat the Raft authors' performance analysis.

Paxos is defined in terms of three abstract agents: the proposers, acceptors and learners. Raft makes no such distinction and only talks explicitly about servers. In Raft, a server can be in one of three states:

Follower A follower is passive and only responds to requests from servers in the other states.

Candidate The candidate state is used to elect a new leader.

Leader A leader handles all client requests and orchestrates the cluster.

Servers communicate using RPCs, and for basic operation Raft only require two RPC types, the RequestVote and AppendEntries RPC. Raft uses a strong form of leadership where the leader drives the communication in a cluster, and other servers only rarely interact with each other.

Raft is designed to work within the crash-recovery model [6]. Under this model, servers can crash and later recover, while still being counted as

correct. For this to work, the servers state must be stored somewhere that persist between failures. Servers in this model will not behave maliciously or fail in arbitrary ways.

2.5.1 Leader Election

As with practical Paxos implementations, the first step in Raft is to elect a leader. A leader broadcasts periodic heartbeats (empty AppendEntries RPCs) to maintain authority. If a follower stops receiving these heartbeats, it will transition into the candidate state and start a new election. In Raft, a new election indicates the start of a *term*. Raft's *election safety* property states that only one server can be elected leader in any given term. A server will vote for a candidate on a first-come-first-served basis, given that:

- The server have not seen a higher term.
- No vote has been granted for an earlier, different, candidate in the same term.
- The candidates log is at least as up-to-date as the voters log.

On becoming a candidate a server will increment its term and start broadcasting RequestVote RPCs to collect votes from the other servers. Eventually one of three things happen:

- The candidate collects a majority of the votes and becomes the leader for that term.
- A response with a higher term or an AppendEntries RPC from a leader will cause the candidate to transition back to the follower state.
- The candidate times out and a new election is started with a higher term number.

Note that a response with a higher term cause the candidate to transition back to the follower state. In Raft, if a server, in any state, receive an RPC with a higher term than its own, that server must increase its term to match and immediately transition to the follower state. As a consequence, if a server is partitioned from the cluster, it can potentially force the leader to step down when the partition is healed. This comes from the fact that the partitioned server will start new elections due to not receiving heartbeats. These elections will have increasing term numbers, typically exceeding the leader's term when a connection to the leader is regained.

To prevent disruptions a pre-election phase is suggested [32]. The pre-election phase is identical to a normal election, except the candidate does not increase its term yet. Instead, the candidate asks the other servers if they would be willing to vote for it in a higher term. A server is willing if the candidate's log is up to date and the server has not recently heard from a leader. Thus, a candidate will not start a real election if it cannot win.

If two ore more followers become candidates around the same time, there is a possibility for a split vote to occur. In the case of split votes no leader is elected for that term. Raft uses randomized election timeouts for every server. This ensures that split votes are uncommon, as followers are unlikely to become candidates at the same time.

2.5.2 Log Replication

After a leader is elected, it can proceed with replicating entries to its followers. This is similar to the second phase of multi-Paxos. As opposed to accept and learn messages in Paxos, log replication is solely driven by the AppendEntries RPC. The AppendEntries RPC also serves as a periodic heartbeat to the followers. When there are no entries to replicate, an empty AppendEntries RPC is issued. This, as we discussed in the previous section, keeps the followers from timing out and starting new elections.

An entry in the Raft log has a unique index and stores a client request, a command which can be applied to the state machine, and the term in which it was added to the log. A command is considered safe to apply to the state machine, *committed*, once the leader has replicated an entry with the command on a majority of the servers.

On receiving a client request, a leader will append the request to its log as a new entry, then invoke AppendEntries RPCs in parallel to the other servers to replicate the entry. After the leader receives confirmation that a majority of the servers have replicated the entry, it can apply the request to its state machine and return the result to the client.

Raft does not allow gaps in the log, as a result, followers will only successfully replicate entries from an AppendEntries RPC that contains entries immediately succeeding its own log. Followers must ensure that the *log matching* property holds before it can apply the received entries:

- If two entries in different logs have the same index and term, then they store the same command.
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

On sending an AppendEntries RPC, the leader includes the index and term of the entry immediately preceding the new entries. Thus, a follower knows that it can safely replicate the new entries if it stores that preceding entry.

Due to the *log matching* property of Raft, a successful round of AppendEntries RPCs commit the entries replicated as well as all the preceding entries. The leader includes the highest index it knows to be committed with every AppendEntries RPC. Thus, followers learn what entries are committed and can apply the corresponding commands to their state machine. This is different from how learners in Paxos learn about which value was chosen. In Paxos learners can learn that a value has been chosen through receiving a learn message for the same value, from a majority of the set of acceptors. This requires a larger number of messages than in Raft. An optimization can be implemented by only sending learns to the leader. The leader then sends a *commit* message to the learners, informing them of the value that was chosen.

During normal operation, the logs of the leader and followers stay in sync. However, leader crashes can cause the logs to become inconsistent. In Raft, the leader is responsible for handling inconsistencies in the log. It does so by telling the follower to duplicate its own log. The leader discovers an inconsistency when a follower rejects an AppendEntries RPC. The leader will then keep sending earlier entries in its log until an AppendEntries RPC is successful. We have reached the point where the leader and the follower started to disagree. Now the leader can start sending entries for increasing indices again, and the AppendEntries RPCs will be successful. This process can be optimized by allowing the follower to respond with the term that it stores for the conflicting entry, and the first index it store for that term. Thus, only one AppendEntries is required for each term with conflicting entries. The Raft authors do however note that they doubt this optimization is necessary in practice.

If a follower crashes and then later recovers, the leader will transfer the log entries that the follower have missed. The number of log entries can be enormous and in some situations it can be preferable to transfer a snapshot of the leader's state instead. In addition, the leader can discard its log up to the point where it took the snapshot. This is a simple form of log compaction called *snapshotting*. The state machine approach makes this possible as only the state is important in the further deterministic execution of the machine.

2.5.3 Reconfiguration

There are two reconfiguration algorithms for Raft presented in Chapter 4 of Ongaro's PhD thesis [32], one for single-server membership changes and another for arbitrary changes. The algorithm for arbitrary changes was introduced with the first publication on Raft [33]. The algorithm for single-server changes came later. As a consequence many of the early Raft implementations implement the algorithm for arbitrary changes. This includes the original Raft implementation *LogCabin* [30]. As all the implementations studied in this thesis use the single-server algorithm, we will not discuss the algorithm for arbitrary membership changes further.

Single-server membership changes

The single-server membership change algorithm for Raft restricts reconfiguration to only one server, i.e., a server can either be added or removed from the cluster at a time. More complex changes has to be performed through a series of reconfigurations. Restricting membership changes to a single server at the time allows a direct switch from one configuration to another. As a consequence, the algorithm is easier to implement than the algorithm for arbitrary changes. The algorithm is safe as a majority in the old and new configuration always overlap in one server, preventing the cluster from splitting into two separate majorities.

When a leader receives a request to add or remove a server from its current configuration, it must first be certain that its current configuration is the latest committed configuration. For a leader to be sure of this, it must first commit an entry in its current term [29]. Once it has been verified that the previous configuration has been committed, the leader appends the new configuration as an entry in its log, and replicates it using the normal Raft mechanism. The new configuration takes effect as soon as it is added to a servers log, and thus a majority of the new configuration is used to determine the commitment of the new configuration's entry. As a result, servers always uses the latest configuration in their log, i.e., they do not wait for the configuration to be committed. The reconfiguration is complete as soon as the configuration's entry is committed. Further configuration changes can now be started.

As configurations are active as soon as they are added to the log, servers must be prepared to fallback to the previously committed configuration in the case a leader overwrites a configuration entry in its log. Thus, a server must keep track of two configurations, the currently committed configuration and the latest configuration stored in its log.

In Raft, it is the calling server's configuration that is used when determining the quorum size, both for voting and log replication. Consequently, servers must accept and process incoming RPCs from servers that are not part of their own current configuration.

When adding a new server to the cluster, the server will usually not store any log entries yet. Given that Raft requires that a server store all the preceding entries before it can replicate new entries, this can cause availability issues. One example is adding a fourth server to a three server cluster. Such a three-server cluster can tolerate one failure, as only two servers are needed to form a quorum. Adding the fourth server increases the quorum size to three but still allows only one server to fail. Moreover, if any other server than the newly added crashes, no new client requests can be replicated until the new server has received all the entries that it is missing. Thus, the cluster is unavailable for the period the new server is catching up. Another way availability loss can happen, is by adding several new servers in quick succession. The quorum size will increase due to the newly added servers, and that will cause the new servers to have to participate in the quorum to make progress. This causes availability loss without any server crashing as the new servers need to be brought up to date before progress can be made again.

In order to avoid this serious availability issue, Raft introduces a phase where the new server joins the cluster only as a non-voting member. As a non-voting member, the server still receives log entries from the leader, but it is not allowed to be part of any quorum for voting or commitment purposes. Once the server has caught up with the rest of the cluster, the reconfiguration can proceed and the server is promoted to a member with voting rights.

It should be noted that this is an issue caused by forcing servers to store the previous entry when voting on the commitment of the succeeding entry in the log. A replicated state machine that builds on Paxos does not have this limitation, as commands can be committed out of order. It does however still require commands to be ordered before applying a command to the state machine. Thus, a server must store all commands to be able to respond to a client request.

To ensure that a replicated service's fault tolerance is not impacted by failed servers, it is desirable to autonomously invoke reconfiguration to replace failed servers. In this scenario, a *window of vulnerability* exists until the reconfiguration is completed [4], as subsequent failures can cause the service to stall forever, requiring manual intervention. This is a well-known problem that has been addressed in several published works [4, 18, 27].

Recall that new configurations are applied as soon as they are added

to the log. As a result, a server that is removed from the configuration stop receiving AppendEntries RPCs immediately. The removed server never learns that it is excluded from the configuration, and as no heartbeats are received, the server will timeout and start new elections with increasing term numbers. Given that any server must allow incoming requests from other servers that are not part of their current configuration, the elections started by the removed server will disrupt the cluster. The current leader is forced to step down due to the higher term number, resulting in poor availability. Raft solves this problem by having followers reject RequestVote RPCs if they have recently heard from a leader.

2.5.4 Batching and Pipelining

There exists two common techniques for improving the performance of replicated state machines, *batching* and *pipelining*. We will explain these concepts from a Raft point of view, however they are fully applicable to other protocols, such as Paxos.

Batching causes the leader to gather client requests until a batch limit or batching timeout is reached. It will then try to reach consensus on all the client request in a single consensus round. This optimizes throughput as it is faster to send one large request than several smaller ones. The same applies when writing to disk.

Pipelining reduces latency under moderate load by optimistically sending AppendEntries RPCs before receiving a response from the preceding ones. Note that the entries in an AppendEntries RPC cannot be committed before the AppendEntries RPCs with the preceding entries have completed successfully, as Raft does not allow gaps in the log.

A thorough investigation of these techniques can be found in [39]. The authors find that the largest gains are to be made by implementing batching, while pipelining is mostly useful in high latency networks.

3
Gorums

Gorums [24] is a new *remote procedure call* (RPC) framework with focus on simplifying the process of building fault tolerant distributed systems. In this chapter we give a thorough introduction to Gorums, as well as highlight some of our contributions to the framework.

3.1 Configurations and Quorum Calls

Gorums allows multiple servers to be combined into *configuration* objects. The configuration objects are immutable and can be any subset of the servers in your cluster. Gorums enables developers to invoke RPCs on the servers in a configuration and wait for replies from a quorum. This is called a *quorum call* (QC).

Quorum calls are exposed as methods on the configuration objects. Invoking a quorum call on a configuration transparently invokes an RPC on each server in that configuration, as can be seen in Figure 3.1. The replies of each individual server that sent a response is not returned by the quorum call. Rather the replies are collected and passed to a *quorum function* which combines them into a single response. A user must provide a quorum function to accompany each quorum call.

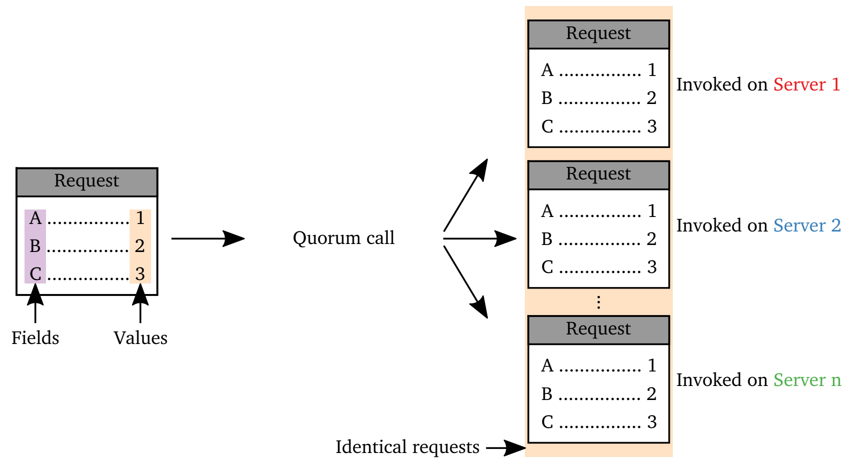


Figure 3.1: Given a single RPC request, a quorum call will transparently invoke the RPC on a specified set of servers.

Algorithm 3.1 Function signature for a quorum function

1: **func** (*qs* QuorumSpec) MyQF(*req* Request, *replies* [ ]Response) Response, Bool

3.2 Quorum Functions

A quorum call waits for a quorum of replies and then returns a single response. To determine when and what to return, a quorum call will execute a user-defined quorum function each time a new response is received. The quorum function have access to a *quorum specification*, the initial RPC request, and the list of replies received so far. This can be seen in Algorithm 3.1 which contains a typical quorum function signature. The quorum specification object or QuorumSpec is used to specify the quorum system. Typically the specification only needs to contain a single parameter defining the quorum size.

The execution of a quorum function is visualized in Figure 3.2. If a single meaningful response can be formed, the quorum function returns the response to the quorum call and the quorum call terminates. On the other hand, if the quorum function requires more information, the quorum call is told to await further replies. If not enough replies are received within a reasonable time, the quorum call will terminate but will include the response from the last execution of the quorum function. The user is informed that no quorum was reached but might chose to act on the partial response constructed.

As indicated by the name, a quorum call is most useful when your procedures need confirmation from a quorum of servers, and one meaningful

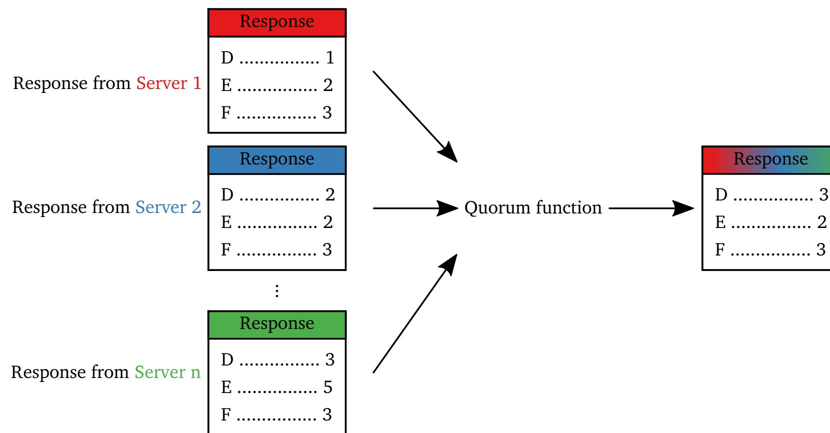


Figure 3.2: As replies to a quorum call are received, the list of replies collected so far is provided as an argument to the user-defined qurom function. The quorum function is responsible for merging the replies into a single meaningful response. Notice that the single response from the quorum function is a combination of the different server responses.

Algorithm 3.2 Example Quorum Function

```
1:  func (qs QuorumSpec) ExampleQF(req Request, replies []Response) Response, Bool
2:      success := 0
3:      for r := range replies do                        ▷ count successful replies
4:          if r.Success then
5:              success++
6:      if success < qs.QuorumSize then
7:          return nil, false                            ▷ no quorum yet, await more replies
8:      reply := new Response                            ▷ initialize reply with nil/0 fields
9:      reply.Success = true
10:     return reply, true                               ▷ quorum found
```

response can be constructed from a quorum of replies. An example of a simple quorum function is shown in Algorithm 3.2. The parent quorum call have sent an identical piece of data to every server in the configuration. To ensure that the data is safe even if some servers fail, the data must be stored on at least a quorum of servers. Each time a new response is received the quorum function will be executed. For each response, the quorum function will check if it were successful (L4) and increment a counter (L5) if so. If the count is less than the quorum size (L6), the quorum function returns indicating that it awaits more replies (L7). Once a quorum is reached, a single successful response is created (L8-L9) and returned (L10).

Algorithm 3.3 Function signature with custom response type enabled

1: **func** (*qs* QUORUMSPEC) MyQF(*req* REQUEST, *replies* []RESPONSE) CUSTOMRESPONSE, BOOL

3.3 Modifiers

Several options are available that modifies the behavior of a quorum call to meet more specific needs.

3.3.1 Custom Response Type

A quorum call requires that a quorum function follow the signature specified in Algorithm 3.1. Notice that the response type for this function is the same as that of the *replies* input argument. This is an implementation detail of Gorums. A quorum call wraps an existing RPC, the one that is invoked on each individual server in a configuration. The RPC defines a specific response type that is the only type available for Gorums to use in its quorum function signatures.

Having the same response type for a quorum call as for the individual RPC is desirable if you are just waiting on a quorum of replies, and the merging consists of selecting and combining field values of that type. In contrast, this can be limiting and excessive if you are actually interested in metadata about the replies. For example, if we only need to know if a quorum of responses were successfully received, a single boolean field is sufficient. Any further details creates unnecessary cognitive overhead for developers. Furthermore, as we are limited to the fields defined in the response type, there is no way to return information that does not fit into that type.

A workaround to the problem of limited fields is to add the required fields to the response type, and let these fields be filled by the quorum function. To illustrate, if we want to know the exact set of servers that responded successfully, we can add an extra list to our response type. However this complicates the design of our system and incurs additional bandwidth as servers now need to send a larger data structure.

We contribute a *Custom Response Type* option to Gorums. This option allows a custom response type to be specified that is different from the RPC response type. The signature of the quroum function is changed to that of Algorithm 3.3. This is visualized in Figure 3.3. Notice how the fields in the response on the right is different from those on the left, as opposed to in Figure 3.2, where they are identical.

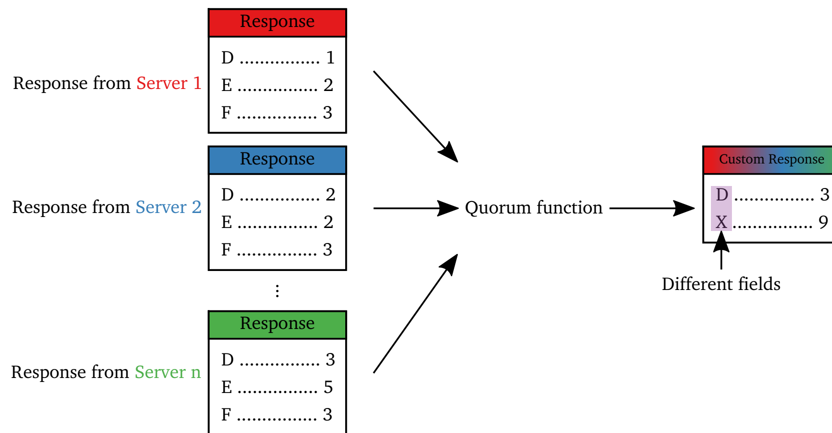


Figure 3.3: Custom response types allow users to define a response type that is different from that of the individual responses. As a result, code can be made easier to reason about by removing excessive fields. Furthermore, additional fields can be added to store metadata that does not fit in the original response type.

3.3.2 Per Server Map Function

A traditional quorum call sends an identical request to each server. This works if we are trying to get the servers to decide on some value. There are, however, cases where it is desirable, in some degree, to control what is sent to each individual server. The initial motivation was dividing data across the servers in a cluster, e.g., an erasure coded storage [1, 38], another is retransmission of missing information that some servers might have lost.

To provide this behavior quorum calls can be modified to run a *per server map function* before sending the request to each server. We have contributed to the design of this option, in particular the use of a base request. The map function is similar in nature to the quorum function but instead of reducing multiple replies into a single response, it creates a unique request for each server. It works by taking the request given to a quorum call, and for each server invoking a user-defined function that modifies that request before sending it to the server in question.

Figure 3.4 shows an overview of how the per server map function works. The base request is modified based on information about the individual servers, this can be seen by the requests having different values.

3.3.3 Correctables

Gorums provide an implementation of the Correctables [13] abstraction. Correctables returns fast, possibly inconsistent responses, as well as a final

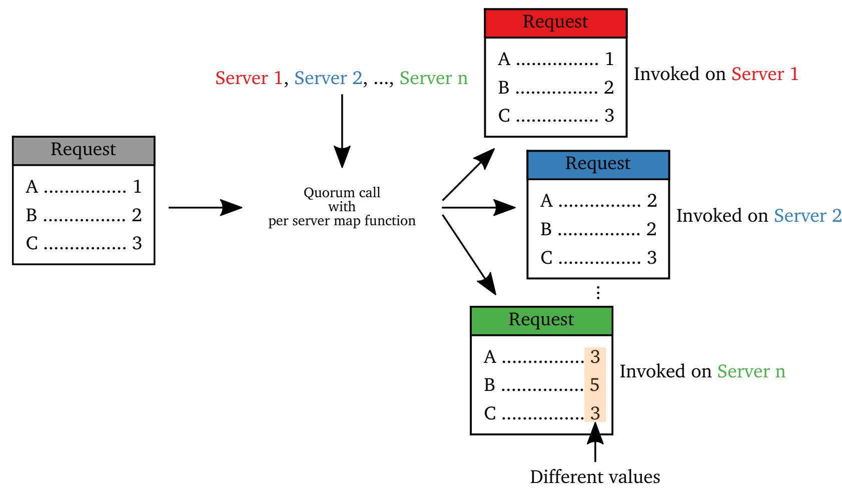


Figure 3.4: A quorum call with the per server map function enabled takes a base request and a user-defined map function. The function is invoked for each server in the configuration creating a modified version of the request. The modified request is then used for the RPC issued to that server.

consistent response. This allows the creation of latency efficient applications that are eventually consistent.

The option modifies a quorum call to not only return a single response but a stream of responses that are incrementally more consistent. To indicate different levels of consistency, the quorum function is changed to return a consistency level with each response. When a higher consistency level is reached, the response is sent on the stream to the caller.

3.4 Implementation Details

Although the abstractions presented in Gorums are language agnostic, there currently only exists one implementation [25], written in the Go programming language [41]. As such, the entirety of the programming work done for this thesis is written in Go. Gorums works by generating Go code based on a description of the services and message types needed for an application. For code generation Gorums depend on Google's *Protocol buffers* [12] and *gRPC* [11].

Protocol buffers is a language-neutral way of efficiently serializing structured data. The data structures that you want to serialize is specified in a *proto* file. The protocol buffer compiler can then be used to generate language specific code, defining the data structures, and how to serialize and deserialize these data structures. As such, programs that

are written in different programming languages can communicate, as they can both read the serialized data.

gRPC allows defining services on the data structures defined in a proto file. These services are also defined within proto files. When the proto file is compiled with the gRPC plugin enabled, code with RPCs for the services are generated.

While being backwards compatible, Gorums extends the service definition ability of gRPC to allow declaring services that work on a set of servers instead of a single server. Developers are still, as for regular gRPC implementation, responsible for implementing the server-side RPC methods. Gorums will generate code that wraps the client-side generated gRPC code.

3.4.1 Cancellation

A gRPC RPC can be explicitly canceled at any time. A cancellation immediately terminates the RPC. This can be applied to avoid needless processing, e.g., if we are invoking redundant RPCs [10]. When a response is received, any outstanding RPCs can be canceled. Gorums extends the notion of cancellation to quorum calls. After a quorum of replies are received, all outstanding RPCs are implicitly canceled.

4 Survey of Raft Implementations

To gain insight into how Raft is implemented in practice, we perform a background study of existing Raft implementations. This will also serve to help us in making good design choices for our own implementation. We start by presenting the works that we have studied, and why those were chosen. Thereafter the implementation of reconfiguration, in these implementations, are analyzed for similarities to themselves and Gorums. Finally, we try to identify some common patterns in the implementations, comparing them with the Gorums abstractions.

4.1 Raft Implementations

Due to the recent popularity of Raft, there exists a multitude of Raft implementations in various stages of development. A table of known implementations with source code available can be found at the Raft website [31]. At the time of writing, *etcd/raft* [8] and *hashicorp/raft* [15] are the two most actively developed implementations written in Go. Furthermore, they are widely used and it is therefore critical that they both perform well and correctly. This, in addition to being written in Go, makes them good candidates for comparison against our Raft with Gorums implementation, *gorums/raft*. Besides they both implement the single-server membership change algorithm described in Section 2.5.3, which we have chosen to implement as well. Lastly, both implementations are mainly used for configuration storage and service discovery. We develop our own implementation for the

same purpose. Thus, we can expect the implementations to behave and perform similarly during evaluation.

4.1.1 etcd/raft

etcd/raft is the Raft implementation backing *etcd* [7], which is a distributed key-value store developed by CoreOS [9]. etcd/raft implements the Raft algorithm as a library, claiming to be the most widely used Raft library in production [8]. To make the behavior of the Raft library deterministic and to ease testing, the Raft algorithm is implemented as a state machine. Modifications to the state machine is done by giving it inputs, which is either an internal message to indicate that time has passed, or receiving a message over the network. In response to an input, an output is generated, and all machines with the same state generates the same output from the same input. The library's output consists of outgoing messages, log entries and changes to the Raft state. Thus, wiring together a cluster of these state machines produces a working Raft cluster.

4.1.2 hashicorp/raft

hashicorp/raft is a core building block in *HashiCorp*'s [16] cloud infrastructure technologies. It is mainly used in *Consul* [14], which provides similar functionality to CoreOS's etcd. It does however come with a large set of builtin features like service discovery and health monitoring. There exists several branches of hashicorp/raft as it is going through a larger rewrite. We examine the branch currently in use by Consul as it must be production ready. hashicorp/raft is less flexible than etcd/raft as a library. etcd/raft requires developer to write the code that allows multiple raft instances to communicate with each other and external systems. With hashicorp/raft developers only need to write the application specific code, i.e., implement the state machine. Everything else is handled by the hashicorp/raft library internally. This, as we mention, makes hashicorp/raft less flexible compare to the etcd/raft implementation. It does, however, shield developers from breaking safety guarantees due to incorrect implementation, and it is a lot easier to work with.

4.2 Reconfiguration

Both etcd/raft and hashicorp/raft implements the single-server membership change algorithm described in Section 2.5.3. This algorithm is much

simpler than traditional reconfiguration algorithms as it allows only one server to be added or removed from the cluster at the time. Yet we find evidence that implementing this algorithm correctly is harder than it looks. A description and solution to a bug discovered in the algorithm have been published [29]. This bug could lead to safety violations when performing reconfigurations. The solution is simple, the leader needs to commit an entry in its term before applying any configuration changes. Earlier revisions of hashicorp/raft were vulnerable to this bug. In addition it allowed multiple configuration changes to be processed before the first one was committed. These bugs have since been patched. etcd/raft is not affected by this bug, as it implements the algorithm in slightly different way.

Although etcd/raft and hashicorp/raft implement the same algorithm, they do so in different ways. hashicorp/raft stays mostly true to the original description, while etcd/raft takes a novel approach. Surprisingly, both implementations add servers directly as voters. As we have discussed, this opens up the possibility for prolonged unavailability.

4.2.1 hashicorp/raft

hashicorp/raft exposes an *AddVoter* method for adding new servers to the cluster. This method's behavior is documented as, first add the server as a non-voting server that will receive log entries, and when it is mostly caught-up with the rest of the cluster, promote it to a voting server. In other words, it avoids the availability issue detailed in Section 2.5.3. In addition, configurations are made active as soon as they are added to a servers log. Thus, hashicorp/raft implements reconfiguration exactly as described in Ongaro's PhD thesis [32]. Further investigation into the hashicorp/raft source code does, however, reveal that this is not entirely true. The step which is supposed to check whether a server is sufficiently up to date is not implemented, and servers are actually added directly as voters. We could find no documentation of this, obviously unexpected, behavior except a brief comment in the source code, stating that the implementation is not trivial.

4.2.2 etcd/raft

The reconfiguration algorithm in etcd/raft deviates slightly from the original single-server membership change algorithm. The main difference is that configurations are not made active until they have been marked as committed, i.e., the new configuration entry is committed under the old

configuration. This is equivalent, by their own claim [8], in terms of safety as the old and new configurations still overlap.

4.3 Communication Abstractions

Although the Raft implementations are quite similar and share many common abstractions, we will now focus on how the different servers in the cluster communicate. This is because we are evaluating Gorums, which is strictly an RPC framework for cluster communication.

Both hashicorp/raft and etcd/raft implement what we will call a *remote peer* abstraction. This abstraction represents a single remote server and provides methods for communicating with it. This is done through RPCs. In addition, a *commitment* or *progress* abstraction is used to control each server's advancement in terms of log replication.

The remote peer abstraction accomplishes a similar goal to that of Gorums' configurations. The key difference being that a configuration is a collection of servers, as opposed to a single server. As a result, Gorums communicates with a set of servers at a time. A quorum call still only return a single response. This response will only contain information that can be applied to the configuration as a whole. This hints at an inherent trade-off between quorum calls and traditional RPCs. Quorum calls promotes separation of concerns and the potential to reduce code complexity, which might come at the cost of losing details about the individual servers.

We will see in Chapter 6 that a separate replication progress abstraction is not needed with Gorums. The progress of the cluster is entirely decided within a quorum call's quorum function.

5 Design and Methodology

The purpose of this thesis is to explore the Gorums framework, and develop a sense of how capable Gorums' abstractions are with regards to log replication and reconfiguration. In this chapter we attempt to clarify how we design a study around solving this proposition. We begin by addressing what we hope to gain by developing an implementation of the Raft consensus algorithm with Gorums. Next, we discuss how a practical application needs to be built on top of the Raft implementation. Further, we show why this replicated service was extended to support multiple Raft implementations. Finally, an overview of the final system architecture will be discussed.

5.1 Evaluation Basis for Gorums

Before we can make any claims about the practicality of Gorums, we need to develop a familiarity with it. For this task we chose to implement the Raft consensus algorithm. An initial study on Raft Consensus with Gorums [36] was performed ahead of this thesis. A naive implementation of Raft was developed, which this thesis builds on. This initial implementation's method of log replication has several problems that needs to be addressed. These problems mainly revolve around a conflict between quorum calls and how the Raft algorithm requires log replication to be implemented. We will discuss these problems and their solutions in depth in Chapter 7. To summarize, we will develop an implementation of Raft with Gorums [35] that addresses these problems and is extended to support reconfiguration.

As we are primarily interested in showing that Gorums can work in a practical application, we also need to develop such an application. Thus, we create a distributed reliable key-value store, the Raft Key-Value Store [34]. This service is built on top of our Raft implementation. As not to make our evaluation of Gorums entirely subjective, we also study and evaluate two state of the art Raft implementations. These were presented in Chapter 4, and they are also the reason behind making a key-value store, as that is the driving component of both Consul and etcd.

Instead of comparing these implementations in isolation, we take this one step further, and extend our key-value store to work all three Raft implementations. Thus, we have a common frame of reference where we can observe the viability of our Gorums implementation by comparing it with the performance of the state of the art.

5.2 System Architecture

We have mentioned that we want to build a service that can be used to evaluate all three Raft implementations. They key property of such a service should be that it is equivalent in terms of the API exposed from the application client's point of view, no matter which Raft implementation is actually being used. This allows the development of an application client designed specifically for testing the system. Such a client would be able to make requests to modify the internal state machine, in this case insert and lookup values based on a key, as well as execute reconfigurations. Performing a series of different operations with this client, on the cluster, will give us information about the various aspects of the cluster's performance. Since the service should be indifferent to which Raft implementation is used, the same series of operations can be run for each Raft implementation. Thus, we can evaluate the different underlying Raft implementations in the same environment, giving us a fair baseline for comparison.

An overview of the system that we have developed to accommodate these requirements are shown in Figure 5.1. The focus of this figure is what we call the application server. This server is what the application client communicates with, it is a single binary containing application logic and a backing Raft implementation for fault tolerance and consistency. As can be seen, clients share the same protocol for communication no matter which Raft implementation is currently in use. The Raft part of the server communicates with other replicated servers through a Raft protocol. The protocol is unique to the Raft implementation in use. This is done intentionally, as it is part of what we are interested in evaluating. That is,

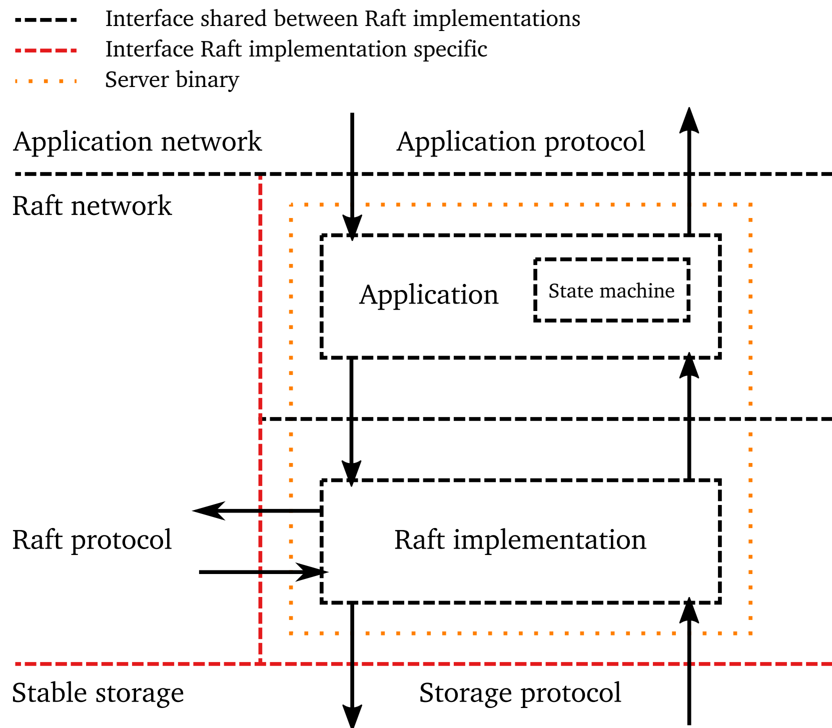


Figure 5.1: High-level overview of the replicated state machine. The underlying raft implementation can be changed without having to alter the rest of the system. This allows different implementations to be compared and evaluated within the same frame of reference.

Gorums is a RPC framework and we want to compare it with the communication abstractions within the other implementations. Storage is another module that is exclusive to the Raft implementation. It would be desirable, and quite possible, to create a common storage solution. In this thesis we decided against doing this due to the storage implementations already being very similar. In the common case, log entries are written to disk in batches in a linear fashion, no random access. In addition, new entries are buffered in memory, so there is little need for reading the data back from disk. To summarize, all the Raft implementations write to disk before issuing or responding to RPCs, and that is what is important.

6
Implementing Raft with Gorums

This chapter presents the implementation of the Raft consensus algorithm with Gorums. We first introduce the changes that need to be made to Raft to accommodate our use of Gorums. Next, we present the leader election process. Furthermore, log replication is described. We finish with the implementation of reconfiguration.

We include algorithms in pseudocode inspired by the Go programming language. These codes are simplified representations of the actual Go code, and provide an overview of the core of our implementation. We take the liberty to hide details where it distracts the purpose of the algorithm, usually we will insert an appropriately named function in their place.

6.1 Changes to Raft

Raft was originally described [33] with traditional one-to-one RPCs as the way of communicating between servers. It follows that the Raft algorithm expects to receive messages from individual servers. When a Raft candidate initiates a request for votes, it needs to send a message to every server, and for every reply received, check whether that constitutes a quorum or if it should abort. When replicating entries to its followers, it does the same thing.

With Gorums, we need to make some changes to Raft. By issuing a single quorum call, we are invoking RPCs on multiple servers at once and receiving a single response. Thus, Raft must be changed in such a way that it expects to send and receive messages that are concerning the whole

cluster. As long as a quorum of replies are received, progress can be made. With Gorums we can go as far as to completely eliminate Raft's need to know about the existence of individual servers in the common case. Raft will receive a single reply from invoking a quorum call. Through this reply, Raft will learn whether the RPCs invoked were successful on a quorum of servers or not.

To summarize, we will remove the need for Raft to perform the individual RPCs to each server. To achieve this, we replace the RequestVote and AppendEntries RPCs with quorum calls.

6.2 Leader Election

Leader election is a central part of the Raft algorithm. It is responsible for electing the leader that will handle client requests and coordinate log replication. We begin by explaining the RequestVote RPC that is invoked on the receiving servers by a candidate. This RPC is implemented as it would have had to be implemented without Gorums. We will then introduce the quorum function which is executed for each RPC response received.

These functions reference and modify the Raft state:

Term is the servers current term.

HaveLeader is true if the server have recently been contacted by a leader.

HaveVoted is true if the server have voted in the current term.

VotedFor contains the candidate voted for in the current term.

Log is the servers log.

6.2.1 RequestVote RPC

The RequestVote RPC can be seen in Algorithm 6.4. It is an implementation of the algorithm as presented in Ongaro's PhD thesis [32], extended to support the pre-election phase introduced in Section 2.5.1. The pre-election phase is an additional election round, used as a test run to verify if the cluster would actually elect the candidate. This prevents the candidate from disrupting the cluster by increasing its own term and starting another election, if it learns that it cannot win.

If a server receive a request from an old term, the server will reject that request (L3). The server's own term is included (L4), informing the candidate that it should cease its efforts to become the leader. Continuing,

Algorithm 6.4 Raft RequestVote RPC

```
 1: func (r RAFT) RequestVoteRPC(req REQUEST) RESPONSE
 2:     reply := new RESPONSE                        ▷ initialize reply with nil/0 fields
 3:     if req.Term < r.Term then
 4:         reply.Term = r.Term
 5:         return reply                             ▷ inform candidate about higher term
 6:     if req.Term > r.Term ∧ ¬req.PreVote then
 7:         r.StepDown(req.Term)          ▷ step down if candidate in higher term unless pre-vote
 8:     reply.Term = r.Term
 9:     if req.PreVote ∧ (r.HaveLeader ∨ (r.HaveVoted ∧ req.Term == r.Term)) then
10:         return reply                  ▷ do not vote for potentially disruptive candidate
11:     votedForCandidate := r.VotedFor == req.CandidateID    ▷ already voted for candidate?
12:     laterTerm := req.LastLogTerm > logTerm(len(r.Log))
13:     sameTerm := req.LastLogTerm == logTerm(len(r.Log))
14:     longEnough := req.LastLogIndex ≥ len(r.Log)
15:     upToDate := sameTerm ∧ longEnough                      ▷ log is up-to-date with ours?
16:     canVote := ¬r.HaveVoted ∨ votedForCandidate ∨ req.PreVote   ▷ restrictions on voting
17:     reply.VoteGranted = canVote ∧ (laterTerm ∨ upToDate)        ▷ vote granted?
18:     if ¬reply.VoteGranted ∨ req.PreVote then
19:         return reply                  ▷ return early if not granted or granted pre-vote
20:     r.Persist(r.VotedFor, req.CandidateID)          ▷ persist voted for to stable storage
21:     r.ResetTimeouts()                              ▷ granting a vote counts as a heartbeat
22:     return reply                                   ▷ vote for candidate
```

Algorithm 6.5 Raft logTerm function

```
1: func (r RAFT) logTerm(index INTEGER) INTEGER
2:     if index < 1 ∨ index > len(r.Log) then
3:         return 0                                    ▷ term 0 if index not in log
4:     return r.Log[index].Term                        ▷ actual term if index in log
```

if a server observers a higher term, the server enters the follower state in that term (L7). This is done using a *StepDown* function which increases the server's term to match the new term. In addition, the server is forced to transition to the follower state. Finally, the follower is given the ability to vote in the new term. This information is persisted to stable storage. Thus, the server does not forget who it voted for if it crashes and then recovers at a later time.

Not storing this information could result in a double vote and violate the election safety property. To prevent potential cluster disruptions, a server checks whether the request is for the pre-election phase (L9). If it is, the server does not grant a vote when it already has a leader, or if its vote was previously spent the requested term.

Raft's *leader completeness* property only allow us to vote for a leader that

Algorithm 6.6 Raft RequestVote Quorum Function

```
 1: func (qs QuorumSpec) RequestVoteQF(req Request, replies []Response) Response, Bool
 2:     reply := new Response                        ▷ initialize reply with nil/0 fields
 3:     if replies[len(replies) − 1].Term > req.Term then
 4:         return reply, true                       ▷ abort if follower in higher term
 5:     votes := 0
 6:     for r := range replies do
 7:         if r.VoteGranted then                    ▷ count votes
 8:             votes++
 9:     if votes ≤ qs.QuorumSize then
10:         return nil, false                        ▷ no quorum yet, await more replies
11:     reply.VoteGranted := true
12:     return reply, true                           ▷ quorum found
```

has a log that is up-to-date with a majority of the cluster. We verify this by ensuring that the candidate's log is more up-to-date than our own, granting the vote only then. A log is defined as more up-to-date if the log have a higher term for the last entry in its log (L12), if the terms are equal (L13), the longer log is more up-to-date (L14). The term of an entry is computed using the *logTerm* function from Algorithm 6.5. An entry not in our log is defined as being in term 0. Furthermore, we can only vote for the candidate if we did not previously grant the vote to another candidate in the current term (L16). If the request is for the pre-election phase, the candidate have the possibility to win the next election. We can thus grant the vote. Now, if we do not grant the vote or this is merely a pre-vote request, we respond to the candidate (L19) with this information.

When we grant a vote for the real election, we need to persist the candidate we voted for to stable storage (L20). Finally, we must reset any timeouts because voting for a candidate counts a heartbeat (L21).

6.2.2 RequestVote Quorum Function

Overall the implementation of the RequestVote quorum call was straightforward. Algorithm 6.6 presents the corresponding RequestVote quorum function that we have developed. First we check whether the latest reply contains a higher term, in which case we abort (L4) by returning the reply initialized to no vote granted. If we do not encounter a higher term, we know that the follower is now in the same term as that of the request. We iterate over the replies counting the number of votes granted (L6-L8). If we have not yet received a quorum, we wait for more replies (L10). In the case a majority of the servers grant a vote, we have received a quorum and

can return successfully (L12).

6.3 Log Replication

The core of the Raft algorithm lies within the AppendEntries RPC. After a leader has been elected, the common case operation of the cluster will revolve around the leader invoking the AppendEntries RPC on its followers. Thus, the design and implementation of this RPC is crucial. Replicating log entries in Raft with Gorums while retaining availability have proven to be one of the most challenging aspects of this thesis.

This is mainly due to the many responsibilities of Raft's AppendEntries RPC. The reason for this comes from Raft being designed in a monolithic fashion to reduce the number of different RPCs needed. We believe this is a breach of the separation of concerns design principle, as we have argued in [36]. As an example, Viewstamped Replication [26] defines at least five message types that are related to log replication, where as Raft encapsulates the whole process into the AppendEntries RPC. We will see this complexity reflected when we present the implementation of the AppendEntries RPC.

In addition, Raft does not allow gaps when replicating log entries. This removes the need for a state transfer phase during leader changes, e.g., Paxos' promise messages as discussed in Section 2.4. In Raft, if a leader is elected, that leader is guaranteed to have an up to date log. Not allowing gaps is however a major problem when using a quorum call for the AppendEntries RPCs. The reason being that individual followers will stall if an AppendEntries RPC is lost, or arrives out of order.

We will now present the AppendEntries RPC, the procedure executed by the leader on every follower. Then we will look at the corresponding quorum function. Finally, we introduce an extension to the Raft algorithm that is intended to address the problem of individual followers stalling on omission faults.

In addition to the state variables in Section 6.2, the AppendEntries related functions introduce:

LatestConfIndex is the index of the active configuration.

CommitIndex is the index of the highest committed log entry, every previous entry is committed.

Algorithm 6.7 Raft AppendEntries RPC

```
 1: func (r RAFT) AppendEntriesRPC(req REQUEST) RESPONSE
 2:     reply := new RESPONSE                                  ▷ initialize reply with nil/0 fields
 3:     if req.Term < r.Term then
 4:         reply.Term = r.Term
 5:         return reply                                       ▷ inform old leader about higher term
 6:     if req.Term > r.Term then
 7:         r.StepDown(req.Term)                               ▷ step down due to higher term
 8:     reply.Term = r.Term
 9:     r.SetLeader(req.LeaderID)                              ▷ acknowledge leader
10:     r.ResetTimeouts()                                      ▷ heartbeat
11:     firstEntry := req.PrevLogIndex == 0                    ▷ first entry is always successful
12:     sameTerm := req.PrevLogTerm == logTerm(req.PrevLogIndex)
13:     logMatches := req.PrevLogIndex ≤ len(r.Log) ∧ sameTerm ▷ log matching property
14:     success := firstEntry ∨ logMatches
15:     if ¬success then
16:         r.MaybeCatchup()                                   ▷ send catchup if not recently sent
17:         req.MatchIndex = r.FirstIndexIn(req.Term)          ▷ index of conflicting entry
18:         return reply                                       ▷ failed to replicate entries
19:     index := req.PrevLogIndex
20:     for i := range 1..len(req.Entries) do                  ▷ duplicate leader's log
21:         index++
22:         if logTerm(index) ≠ req.Entries[i].Term then       ▷ check if log entry conflict
23:             if req.Entries[i].Index == r.LatestConfIndex then ▷ overwriting configuration?
24:                 r.RollBackConf()                           ▷ rollback to committed configuration
25:             r.Log = r.Log[1 : index − 1] + req.Entries[i]  ▷ truncate and place correct entry
26:             if req.Entries[i].Type == CONFIGURATION then   ▷ new configuration?
27:                 r.ApplyConf(req.Entries[i])                ▷ apply configuration
28:     r.CommitIndex = min(req.CommitIndex, index)            ▷ update commit index
29:     reply.MatchIndex = index
30:     reply.Success = true
31:     return reply                                           ▷ successfully replicated entries
```

6.3.1 AppendEntries RPC

As with the RequestVote RPC, the AppendEntries RPC in Algorithm 6.7 is similar to how it would have been implemented without using Gorums.

The AppendEntries RPC handles term numbers that are either lower or higher than its own, in the same way as the RequestVote RPC. If we make it to L9, it means that this follower is in the same term as the leader calling the AppendEntries RPC. Thus, we acknowledge the caller as the leader and count this AppendEntries RPC as a heartbeat (L10). Further, we need to identify whether the entries that we have received can be placed in our log. This is always possible if we receive entries directly succeeding index 0 (L11). On the other hand, we need to make sure that we store the entry

immediately preceding the entries received (L13). It then follows from the log matching property of Raft, that our logs are identical to the leader up to and including that entry. It is thus safe to replicate the received entries.

If we are not successful in replicating the entries, we calculate the index of the conflicting entry (L17) and inform the leader (L18). If the leader processes this response, the index sent is used in finding out what entries the follower needs to be successful. In addition, we send a Catchup RPC to the leader (L16). This addresses a problem where the leader ignores to processes the response from this RPC, and is further explained in Section 6.3.3.

If we do not return due to being unsuccessful (L18), we have established that we store the entry immediately preceding the entries received (L13). Thus, we append the new entries after what is specified as the previous entry's index (L19). In some edge cases there might be log entries in our log past this index. These are safe to overwrite (L25), as they are not committed.

Finally, we inform the leader that we have successfully replicated the entries (L31).

We discuss the parts of the algorithm that apply to reconfiguration (L23-L24, L26-L27) in Section 6.4.

6.3.2 AppendEntries Quorum Function

Algorithm 6.8 shows the AppendEntries quorum function. This function is similar in structure to Algorithm 6.6, the RequestVote quorum function. It aborts when a higher term is seen (L5), waits for more replies when there is no quorum (L15), and it constructs a single reply and returns successfully when there is a quorum (L18). If a RequestVote RPC fails, we just send a new one with a higher term number. This is not possible if the AppendEntries RPC fails, as we need to back off and send the missing entries first. When a follower can not apply the received entries it responds with the index of the conflicting entry. Thus, we need to keep track of this index as we process replies. As a quorum call acts on the cluster as a whole, we need to find the lowest index (L11). This allows us to construct the next AppendEntries request such that it will be successful on the followers who responded. Note that this only happens if a majority of the servers respond unsuccessfully. If we cannot get a quorum of successful replies, we wait until every server have responded or the RPC times out. Thus every operational server should have responded. If a quorum of successful responses are received, the tracked index is replaced to reflect that every entry have been replicated (L16), and that new entries succeeding this index can be

Algorithm 6.8 Raft AppendEntries Quorum Function

```
 1: func (qs QuorumSpec) AppendEQF(req Request, replies []Response) CustomResponse, Bool
 2:     reply := new CustomResponse                       ▷ initialize reply with nil/0 fields
 3:     reply.Term = replies[len(replies) − 1].Term       ▷ term of latest reply
 4:     if reply.Term > req.Term then
 5:         return reply, true                            ▷ abort if follower in higher term
 6:     reply.Replies = len(replies)                      ▷ note how many responded
 7:     reply.MatchIndex = ∞
 8:     successful := 0
 9:     for r := range replies do
10:         if r.MatchIndex < reply.MatchIndex then
11:             reply.MatchIndex = r.MatchIndex           ▷ set lowest match index
12:         if r.success then                             ▷ count successful replies
13:             successful++
14:     if successful ≤ qs.QuorumSize then
15:         return nil, false                             ▷ no quorum yet, await more replies
16:     reply.MatchIndex = req.PrevLogIndex + len(req.Entries)   ▷ restore match index
17:     reply.success := true
18:     return reply, true                                ▷ quorum found
```

sent.

6.3.3 Raft Catchup Extension

Raft requires that a follower store all the preceding log entries to participate in replicating and committing successive entries. Thus, a follower that is missing some entries cannot make progress until those entries have been received. Our AppendEntries quorum call is not able to handle the instance where individual followers fail to replicate entries, only when at least a quorum of followers do. This is because the quorum function returns a failed response including the lowest match index. Thus, steps must be taken to back off and replicate the missing entries. There is no guarantee that this will bring all followers up to date, only a quorum. To address this problem we develop a *Catchup extension* for Raft. We explain in detail the need for this extension and alternatives in Chapter 7.

The extension allows for followers to inform their leader about which point in the log they are stalled. This is done with a Catchup RPC. When the RPC is invoked on a leader, the leader stores the index contained within the Catchup RPC until a new AppendEntries quorum call is due. Knowing how far behind a follower is, it piggybacks all the missing entries on the next AppendEntries RPC to that follower. The AppendEntries RPC will also contain the new entries immediately following the missed entries. Thus, a

Algorithm 6.9 Raft AppendEntries Per Server Map Function

```
1: func AppendEntriesRequestMapper(req REQUEST, serverID INTEGER) REQUEST
2:     if index, ok := receivedCatchup(serverID); ok then        ▷ recevied catchup from server
3:         req.PrevLogIndex = index − 1     ▷ change previous entry to match catchup request's index
4:         req.PrevLogTerm = r.logTerm(index − 1)                  ▷ update the previous entry's term
5:     numEntries := maxIndex − req.PrevLogIndex
6:     req.Entries = entries[total − numEntries : total]   ▷ only send include numEntries entries
7:     return req
```

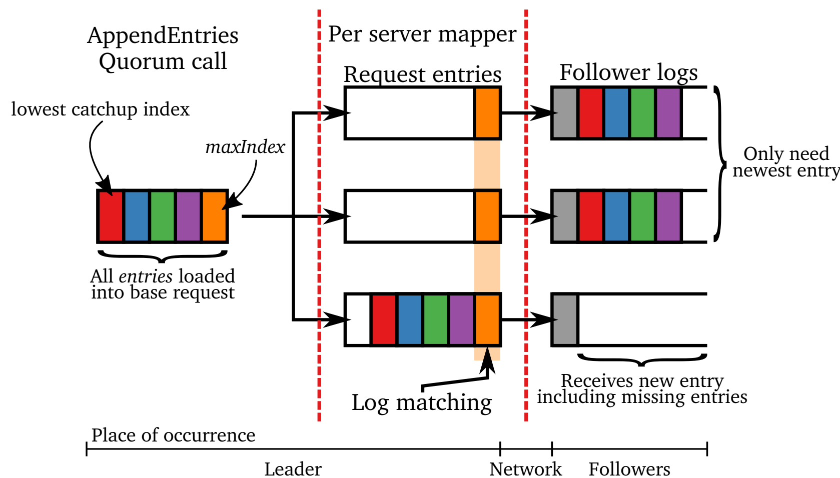


Figure 6.1: When a leader receives a catchup request, it includes the missing entries with the next AppenEntries RPC in addition to any new entries. This causes a follower that was behind to be able to participate in committing the latest entries. The colored rectangles represent log entries.

follower that lost some entries, can still participate in voting on the commitment of the new entries. This property is critical, as without it, we would have had to implement additional mechanisms to handle followers lagging behind.

The Catchup extension is made possible through the per server map function introduced in Section 3.3.2. This function allows us to manipulate the request to each follower, such that it contains all the entries required by each follower. The function is presented in Algorithm 6.9. The variable *maxIndex* holds the index of the last entry that we plan to replicate the current AppendEntries quorum call. All the log entries from the lowest catchup index received to *maxIndex* is made available in *entries*. These are already cached in memory. If a follower is further behind than the cache allows, it cannot catchup given the current catchup implementation. In which case a reconfiguration would need to be performed to replace the stalled follower. We experiment with replacing faulty servers in Section 8.5.

If a catchup is received, the map function will replace the previous entry description with the description of the entry immediately preceding the index received (L3-L4). The entries following this entry is then placed within the RPC request (L6). Thus, the follower executing this RPC is able to replicate and vote on the commitment of the entries received.

A visualization of the AppendEntries quorum call with the Catchup extension enabled is shown in Figure 6.1. First the leader creates a base request where it includes all the required entries. Then three copies of that request is made and modified to reflect the log entries each individual follower requires. The important part being that every follower receive the latest log entry. Thus, when the leader receive a successful response, it knows that the follower stores all entries up to and including the latest entry.

6.4 Reconfiguration

For reconfiguration, we implement the single-server membership changes algorithm, as discussed in Section 2.5.3. This algorithm makes use of the fact that transitions between configurations always overlap in their majorities. As a result, the algorithm only requires a little additional mechanism on top of the already established log replication implementation. To put it another way, the reconfiguration algorithm is tightly coupled with the AppendEntries RPC.

The leader applies a new configuration immediately after appending it to its own log. Followers learn about this configuration through the AppendEntries RPC. The configuration is applied when it is added to the follower's log, as can be seen in the previously discussed Algorithm 6.7 (L25-L27). As the leader is potentially overwriting entries in the follower's log (L25), the follower must be ready to fall back to the previously committed configuration (L23-L24).

We have mentioned that an additional phase is needed to avoid availability gaps after reconfiguration. This is where the complexity of the algorithm lies. First, the leader must transfer the log entries that the follower is missing. This needs to happen in parallel to processing new requests from clients. Waiting for the transfer to complete will work, but cause latency issues. In addition, we need some way to determine when a follower is sufficiently up to date, so that it can be safely included in cluster operations.

There is already a mechanism for replicating entries to a follower, the AppendEntries RPC. Thus, we can create a configuration that contains only the follower that we want to bring up to date. We can then invoke Append-

Entries quorum calls on this configuration to transfer the missing entries. This is done at a higher rate than with the regular AppendEntries quorum call on the main configuration. As a result, the follower will eventually catch up with the rest of the cluster. Before the leader issues an AppendEntries quorum call on the main configuration, it will check the progress of the follower being brought up to date. If it matches the progress of the main configuration, the leader will create a new configuration that includes the new follower. The leader then appends this configuration to its own log, and the next AppendEntries quorum call is invoked on the new configuration. Hence, the first entries the newly added follower replicates as part of the cluster, will contain the entry that includes itself in the cluster. This entry is used to indicate to the follower that it is now a full member of the cluster, allowing it to time out and start elections. Prior to this point, the follower would only respond to AppendEntries RPCs from a leader. Thus, if a reconfiguration request to add a server where to time out, the request could simply be repeated and log replication will start off where it ended.

7 Implementation Challenges

This chapter presents the challenges encountered when developing an implementation of the Raft consensus algorithm with Gorums.

Raft enforces that log entries be replicated in a strictly linear fashion. Thus, followers are not allowed to acknowledge an AppendEntries RPC with entries that would cause a gap in the followers log. The advantage of this approach is that when electing a new leader, no state transfer is required, as the leader is required to already store all log entries.

This is a reasonable approach as Raft is described with RPCs invoked on each individual follower in parallel. When a follower rejects an AppendEntries RPC, a new one is simply invoked with entries from earlier in the log.

However, using Gorums and its quorum call abstraction, this does not work out so well. We will demonstrate in the following sections, how the forced order on replicating log entries in Raft causes a range of challenges in its implementation with Gorums.

7.1 Canceled Replies

Recall from Section 3.4.1 that a quorum call will implicitly cancel any outstanding RPCs on receiving a quorum of replies. This makes sense from a performance point of view, however in some cases it can have some unfortunate consequences. A quorum call invokes RPCs on individual servers asynchronously. Canceling outstanding RPCs on receiving a quorum can

causes any of these RPCs to never be sent in the first place. This is problematic if you want every server to receive and process the RPC, i.e., if the RPC contains log entries you want replicated.

Raft is especially vulnerable to omission faults as log entries need to be replicated in a strictly linear order. If an AppendEntries RPC is canceled as soon as a quorum is reached, there may be followers that did not receive the log entries for that RPC. This causes all further AppendEntries RPCs to fail for that follower. After a while, multiple followers may stall and only a quorum of followers store all entries. This is an availability problem which cannot be tolerated in any practical system.

We solve this problem by extending Gorums to allow the caller of the quorum call to decide when it is time to cancel. In the case of our AppendEntries quorum call we do not cancel unless we decide to abort. Thus, every AppendEntries RPC is given time to complete, not only a quorum of them. If we abort, it is okay to cancel the RPCs as the caller must step down from the leader state, and is unable to process any remaining replies.

7.2 Ignored Replies

In Raft when a follower responds to a request from a leader, it includes important information about its own state. The two most important pieces of information is whether the request was successful and which term the follower is currently in.

Using Gorums we tend to focus on the state of the majority of servers. This does however mean that we lose track of the progress of individual servers. As long as a quorum of servers respond successfully to an AppendEntries RPC, we ignore any response from a server who did not succeed. As we have discussed earlier, in Raft, we cannot ignore omission faults. If a follower fails one AppendEntries RPC, it will fail all further RPCs until the missing log entries are received. Thus, it is critical for the availability of the cluster that the leader learns about the follower's missing entries.

We just discussed canceled replies, where quorum call replies are lost due to some RPCs never being issued in the first place. There is another way quorum call replies might be ignored. When a quorum call have received a quorum and returned, all outstanding replies will not be processed. Thus, if a quorum of successful replies are received before a failure, the caller never learns about the failed requests. This problem could be resolved by changing the quorum call to wait for a response from every server before proceeding. However, waiting for every server is not practical as we would slow progress down to the speed of the slowest follower or, in the worst

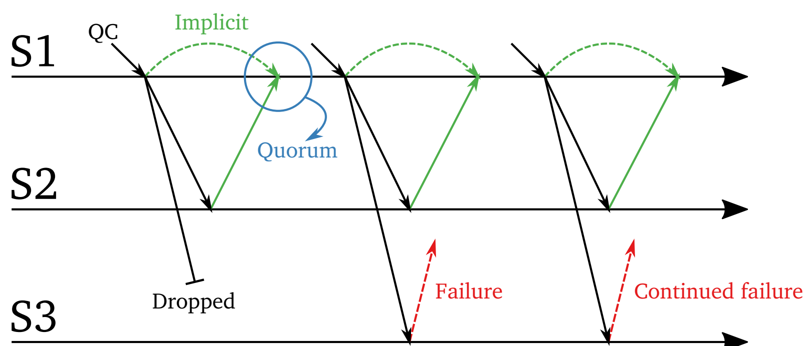


Figure 7.1: S1 is the leader of the cluster issuing AppendEntries quorum calls to replicate log entries. S3 never receives the first AppendEntries RPC. This causes further RPCs to S3 to fail. The leader is never informed of S3 stalling behind, as a response from S2 is enough to form a quorum. Thus, the quorum call always return a successful response, but the cluster availability is now impacted.

case, the quorum call's timeout.

Figure 7.1 depicts how we initially imagined that a failure due to an ignored reply would play out. S1 is the leader, issuing AppendEntries quorum calls. An AppendEntries RPC to follower S3 is dropped. However, the parent AppendEntries quorum call finishes without problems, and reports back to the leader that the entries have been successfully replicated to a quorum of its followers. This is enough for the leader to make progress. In this scenario, S3 receives the following AppendEntries RPC. Since S3 is missing the entries from the previous call, it cannot replicate the new entries yet. S3 will inform the leader of this fact by responding with a rejection of the entries. Meanwhile, the leader have already received a confirmation from S2, including itself, forming a majority. Thus, the quorum call have already returned, and the response from S3 is never processed. The leader never learns that S3 is stalled, and will be unaware of this fact keep sending AppendEntries RPCs with the latest entries that it wishes to replicate.

For starters, this is a huge availability problem. As the leader never learns that a follower have stalled, eventually only a quorum of the cluster makes progress. In addition, the leader keeps sending log entries to a follower that have explicitly stated that it cannot accept these entries yet, wasting potentially large amounts of bandwidth.

When only a quorum of the servers store all entries, a single server crashing will cause a loss of availability. No progress can be made replicating client requests until one of the stalled followers is brought up to date. This is similar to the example about adding a new server in Section 2.5.3.

Figure 7.2: Introducing the Catchup RPC, S3 is able to inform the leader about the point at which it is stalled. The leader is thus able to include the missing entries in its next AppendEntries RPC to that follower.

A reasonable assumption is that server crashes are less frequent than some network packets being dropped. Thus, stalled followers are likely to be so far behind that they require a full snapshot of the state machine to recover.

To handle neglected omission faults we develop the Catchup extension whose implementation we introduced in Section 6.3.3. The Catchup RPC provides followers with the ability to inform the leader when it suspects some log entries might have been lost. The leader is then able to augment its next AppendEntries quorum call to account for the missing entries.

Figure 7.2 shows the previous scenario with the Catchup extension enabled. In this case, S3 notices that it has missed the previous AppendEntries RPC and sends a Catchup RPC to inform the leader. The failed AppendEntries RPC response is still sent to the leader. This can help prevent the leader from stepping down, as discussed in Section 7.5. When the leader receives the Catchup RPC, it augments its next AppendEntries RPC to that follower to contain the missing entries. As a result, we see S3 successfully responding to the latest AppendEntries RPC from the leader.

We believe that the catchup mechanism upholds the safety properties of Raft. This is due to a Catchup RPC only causing the leader to retransmit entries that it has already sent and Raft's RPCs are idempotent.

In terms of liveness, some limitations apply. The current implementation only allows entries that are still cached in memory to be retransmitted. It would not be advisable to read a large number of entries from disk, as this will greatly influence the performance of the cluster. Potentially causing followers to time out. If the entries a follower needs are no longer in the cache, that follower cannot recover using the Catchup RPC.

The Catchup extension could be further extended to handle this situa-

tion as well. The leader could calculate how far behind a follower is with respect to receiving a Catchup RPC. If the log entries are no longer available, a snapshot of the leader's state machine could be sent in its place. Thus, the follower would only need to receive the log entries from after the snapshot was created. We do not implement this behavior.

The main drawback with the Catchup RPC is that it is an unproven extension to the Raft algorithm. This is not something you want to have to do when implementing an already complex system. The upside being that it solves the omission fault issue we describe and it performs quit well, as we shall see in Chapter 8.

7.3 Reordered Requests

During the experiments performed in preparation for Chapter 8, we noticed that an unusual number of Catchup RPCs were being invoked, even under ideal network conditions. Eventually we were able to attribute this strange behavior to a quirk in how Gorums executes the RPCs for quorum calls. We actually first experienced this quirk running a Raft cluster on a single host but wrote the behavior off as not possible in a *local area network* (LAN) or *wide area network* (WAN) setting. Further experiments have demonstrated that this was an incorrect assumption.

In Section 7.1 we introduced the possibility to explicitly cancel a quorum call. By choosing not to cancel an AppendEntries quroum call, the AppendEntries RPC issued to each follower is allowed to complete. This ensures that the log entries sent are replicated on as many followers as possible.

It is important to note here that the quorum call returns on receiving a quorum of successful responses. After the quorum call have returned, there might still be AppendEntries RPCs that have not yet been issued. Not canceling the quorum call allows these to eventually complete.

Raft will perform another AppendEntries quorum call as soon as there are more entries ready to replicate. Now, as a new set of AppendEntries RPCs are issued to each follower. There might still be some AppendEntries RPCs from the previous quorum call that have yet to be issued. It is then possible that the RPCs for the new quorum call are executed before these previous ones.

Figure 7.3 depicts this rather involved issue. It repeats the scenario from Figures 7.1 and 7.2, only this time the AppendEntries RPC to S3 is delayed not dropped. As before, S1 is the leader.

This time we also show the Go routines responsible for issuing the indi-

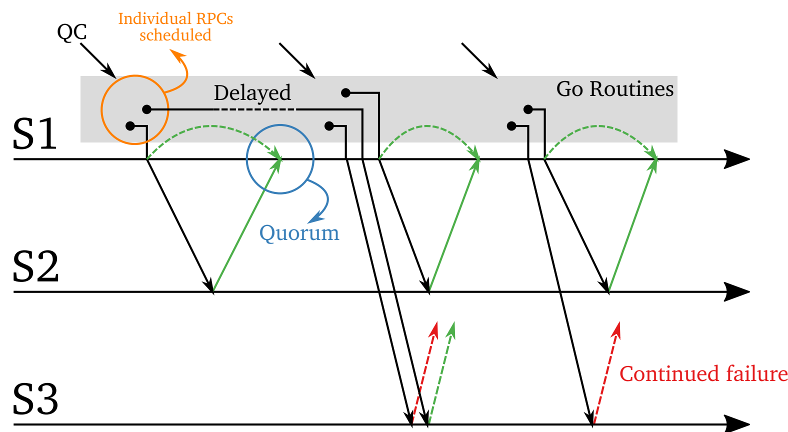


Figure 7.3: An AppendEntries quorum call is invoked, creating a routine responsible for issuing an individual AppendEntries RPC to each follower. The RPC to S2 is successful, and the quorum call returns. A new quorum call is issued, however the RPC to S3 from the previous quorum call have yet to be executed. The RPC to S3 for the new quorum call is executed first, causing the RPCs to be invoked out of order, with respect to the expected invocation order.

vidual AppendEntries RPCs. Go routines can be thought of as threads, they allow concurrent execution of code. The routine responsible for issuing the RPC to S2 is scheduled first. By this we mean the code that delivers the RPC request to the network layer is executed first. The leader then receives a successful response and the quorum call returns. Note that the routine meant for S3 have yet to be scheduled.

Then a new AppendEntries quorum call is invoked by the leader, creating two additional routines for the RPCs to each follower. The routine issuing the newest RPC for S3 is scheduled first. This RPC contains entries that S3 cannot safely replicate, as it have not yet received the previous RPC.

This is a problem, as we have discussed in great detail, because Raft can only replicate log entries where it stores the immediately preceding log entries. Enabling the Catchup extension handles this problem, as can be seen in Figure 7.4. The problem is not solved, the Catchup extension only treats the symptoms.

An interesting detail of this scenario is that due to the second RPC being issued before the first, the *Transmission Control Protocol* (TCP) guarantees that these are invoked out of order. As such, forcing Gorums to issue the RPCs in the correct order will also solve our problem. We are hesitant to call this a bug as some protocols might not require RPCs to be invoked in

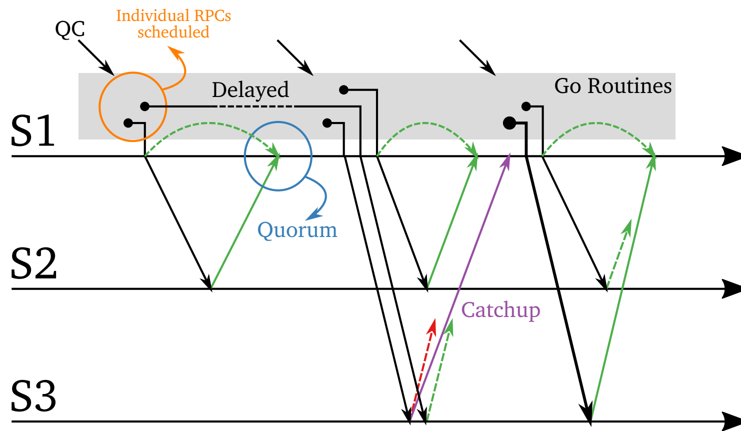


Figure 7.4: Enabling the Catchup extension gracefully handles the case where S3 have two RPCs invoked out of order. The leader is informed and includes the missed entries in the next AppendEntries RPC to S3.

order, as Raft does.

Learning that Gorums can invoke RPCs out of order, we realize that the failure scenario that we describe in Section 7.2 is incorrect. The failure does not happen due to a lost AppendEntries RPC, but because an AppendEntries RPC is received out of order and discarded. Thus, all following AppendEntries RPCs fail.

To solve the problem of reordered RPC invocations we develop the *Strict Ordering* option for Gorums. A quorum call $qc_x(F)$ is defined as $\forall f \in F\colon rpc_x(f)$, where f is a follower in configuration F and $rpc_x(f)$ is the RPC to be issued by $qc_x(F)$ to the follower f. The Strict Ordering option guarantees that if a quorum call $qc_1(F)$ is invoked before another quorum call $qc_2(F)$, the RPC for a follower $rpc_1(f)$ created by $qc_1(F)$ is always executed before the RPC $rpc_2(f)$ for the same follower created by $qc_2(F)$. In short:

$$\forall f \in F\colon rpc_1(f) \rightarrow rpc_2(f)$$
$$\forall f_1, f_2 \in F \text{ where } f_1 \neq f_2\colon \quad rpc_i(f_1) \| rpc_j(f_2) \qquad i, j \in \mathbb{N}$$

- The $\rightarrow$ relation defines that the call on the left is executed before the one on the right.
- The $\|$ relation defines that the call on the left and right are executed independently of each other.

Thus, enabling the Strict Ordering option solves the problem of followers receiving AppendEntries RPCs out of order. The leader is forced to invoked the AppendEntries RPC in the correct order to each individual follower.

7.4 Failure Detection

While a follower can easily tell the leader when it is missing entries, informing the leader of its own crash is impossible. The way our quorum calls are designed, a server failure is completely masked as long as a quorum of servers are still operational. While this is intentional and desirable, it is also important to be informed about potential problems in the cluster. If servers keep crashing and we do not do anything about it, it will not be long until the cluster can no longer make progress. Traditionally we could have used the lack of responses to RPCs or a connection closed event as a way to detect problems. With Gorums we get one response from the configuration we invoked the quorum call on. As such, we cannot make any assumptions about the status of the individual servers. We could use a custom response type to encode this information in the response to a quorum call. However, due to the problem with ignored responses, we would only get information about the servers that replied fast enough. Thus, we cannot differentiate a slow server from a server that is partitioned from the cluster or have crashed. In this case we define slow as responding within the RPC timeout but after the quorum call have returned.

While a quorum call ignores replies which arrive too late, the underlying RPC is still completed. Gorums exposes a way to check if any RPC error has occurred for a specific server. This error can be used to check whether a server is unavailable. We use this approach to detect potential problems in the cluster. Thus, a leader could use this information to perform reconfigurations that replace faulty servers.

7.5 Retaining Leadership

Client requests in Raft are processed by the leader. Consequently, clients need to find the server in the cluster which is the leader. This is done by having the client send its first request to a random server. If that server is not the leader it will reject the request. In addition, the server will include the address to the server that it believes to be the leader. Thus, the client can speed up the process by connecting directly to the leader, or if no address is received, try another random server.

The problem with this approach is that a leader may be partitioned from the rest of the cluster, and still believe itself to be the leader. If clients connect to this leader, the clients' requests might stall forever, as the server is not able to replicate log entries to any other servers. At the same time there might be a leader for a higher term, on the other side of the partition, that is able to commit client requests. Raft solves this problem by forcing a leader to step down if it can no longer maintain a heartbeat to a majority of the cluster.

With our AppendEntries quorum call, a successful response indicates that the leader maintains a connection to at least a majority of its followers. This can be true, even in the case of a failed response. A failed response only indicates that the log entries where rejected, the followers still acknowledge the leader. The exception is receiving a response with a higher term as the leader is forced to step down.

The AppendEntries quorum call does not return the number of servers that replied. Thus, we cannot tell if a majority of servers have in fact received a heartbeat, if the response was unsuccessful. We solve this problem using the custom response type option, in which a field is introduced to contain the number of replies fed to the quorum function. As a result, the number of servers that replied is returned from the quorum call. Thus, if an unsuccessful response is returned, and only a minority responded, the leader must step down as it could not contact a quorum of servers.

It should be noted that there exists a quorum call *future* option that does in fact provide the exact number of servers that responded. At the time of writing, it is however not compatible with the per server map function option. A per server map function is required by the Catchup extension, preventing us from applying this option.

7.6 Correctables

Many of the problems we face with Gorums can be solved with some clever use of the correctables option. One example is dealing with missing entries, were we implemented an additional Catchup RPC to resolve this issue. We could have used correctables to learn this information instead. That is, the quorum call could return as soon as we have a quorum, we could then process the remaining replies as they are received. This could also be used to detect servers that have not responded to that quorum call.

We have chosen not to try implementing these approaches as they are abusing the correctables option by using it for something it was not meant for. In addition, by using correctables our implementation quickly shifts the

focus away from quorums to individual servers. This brings back a lot of the complexity we are trying to avoid in the first place.

8 Experimental Evaluation

As quorum calls in Gorums only wait for a quorum of responses, the followers not participating in such a quorum are particularly vulnerable to omission faults. If a follower loses only a single AppendEntries request, it can no longer make progress. In this thesis we develop three follower recovery mechanisms to handle this situation. First, to handle brief omission faults and reordering of messages, we propose the Catchup extension to the Raft algorithm and the Strict Ordering option for Gorums. Further we suggest a reconfiguration policy which deals with larger failures, such as server crashes and longer partitions. Throughout this chapter we will evaluate these mechanisms using two industry standards for comparison. We will first present our experimental setup and examine the common case operation of a cluster, then we will evaluate the Catchup extension and Strict Ordering option, finally we construct an experiment to assess the reconfiguration of the three Raft implementations.

8.1 Code Coverage

Reliability and robustness are of the highest priority in critical distributed systems. Testing distributed systems is difficult as there a many moving parts and an uncountable number of execution paths. Thus, a well-written test suite, that test the system both in isolation and in practice, is needed to ensure correctness.

One of the benefits of using Gorums is that it promotes separation of concerns which simplifies testing [23]. Table 8.1 shows the code cover-

Table 8.1: Code coverage for the different Raft implementations.

Package	Function	Coverage
github.com/hashicorp/raft	Raft and Networking	80.6 %
github.com/hashicorp/raft-boltdb	Storage	85.0 %
github.com/coreos/etcd/raft	Raft	86.8 %
github.com/coreos/etcd/wal	Storage	79.6 %
github.com/coreos/etcd/rafthttp	Networking	75.9 %
github.com/relab/raft/raftgorums	Raft	84.2 %
github.com/relab/raft	Storage, Utilities, Metrics	43.1 %
github.com/relab/gorums/dev	Networking	44.9 %

age of the most important components of the three Raft implementations we study. Both etcd/raft and hashicorp/raft have good coverage of their Raft algorithm, as well as storage and networking. Our gorums/raft is on par with these, providing both unit and integration tests. The gorums/raft storage is well tested but the numbers are skewed due to utility and metrics code that does not need tests. The Gorums package contains code generated by gRPC and Protobuf, e.g., for serializing and deserializing the RPC data types. This code should already be covered by the tests in these dependencies. Hence, the observed Gorums code coverage is lower than it actually is.

A recent work [43] on model-based testing [42], uses Gorums in its study. The work demonstrates that 100 % and 84 % code coverage can be obtained for quorum functions and quorum calls, respectively.

8.2 Experimental Setup

All experiments are run on virtual machines from Amazon Elastic Compute Cloud (Amazon EC2) [3] within the same data center, i.e., in a LAN environment. We use different availability zones to ensure that the virtual machines are on separate physical machines. Availability zones are isolated locations within one data center. This is to prevent any follower from being placed on the same machine as the leader. Unless otherwise noted, an experiment will always use 200 client processes running on a single server and a Raft cluster size of 5 servers. As there were only 5 availability zones available within our chosen region (us-east-1), we prevent a server from becoming the leader if it is placed in the same zone as the client. Amazon EC2 virtual machines are divided into instance types based on, i.e., CPU, memory, storage, and networking capacity. For the client we use a

Table 8.2: Specification for server and client machines [2].

	Model	vCPU	Mem (GiB)
Server	r3.xlarge	4	30.5
Client	r3.2xlarge	8	61.0

r3.2xlarge instance type and for the Raft servers we use *r3.xlarge* instances. The instance capabilities can be seen in Table 8.2. Both instance types use *solid-state drives* (SSD) for storage. All servers are running the *Amazon Linux AMI* operating system and the code is compiled using Go 1.8.1. The experiments are run with the Go garbage collector disabled to avoid distortion.

All experiments are automated through Ansible [37] playbooks. The playbooks, with minor changes, can be used to recreate these experiments on Amazon EC2, other cloud providers or physical machines. They are made available as part of the Raft Key-Value Store code repository [34].

For the replicated service we use the key-value store presented in Chapter 5. The store is a simple replicated hash map. It uses the string data type for both key and value. Clients are able to issue read and write requests to consistently alter the hash map. In addition, requests can be made to change the configuration of servers making up the replicated service.

Snapshotting is disabled in all Raft implementations. This is done to minimize the number of variables in our experiments. When adding a server to the cluster by means of reconfiguration, the newly added server never stores any log entries. We use batching and pipelining to improve performance, as described in Section 2.5.4. Experiments in this chapter are run with batching enabled on every implementation. Pipelining is enabled in hashicorp/raft and etcd/raft but currently not supported in gorums/raft.

8.3 Common Case Operation

In this first experiment we examine the performance of the different Raft implementations in common case operation. We give the replicated service a stable and constant stream of client write requests. All the Raft implementations implement a client request batching mechanism. This optimizes throughput as it is faster to send one large request than several smaller ones. The same applies when writing to disk. Both hashicorp/raft and etcd/raft implement some form of pipelining as well. This reduces latency under moderate load by optimistically sending AppendEntries before receiving confirmation for the preceding ones. This experiment will

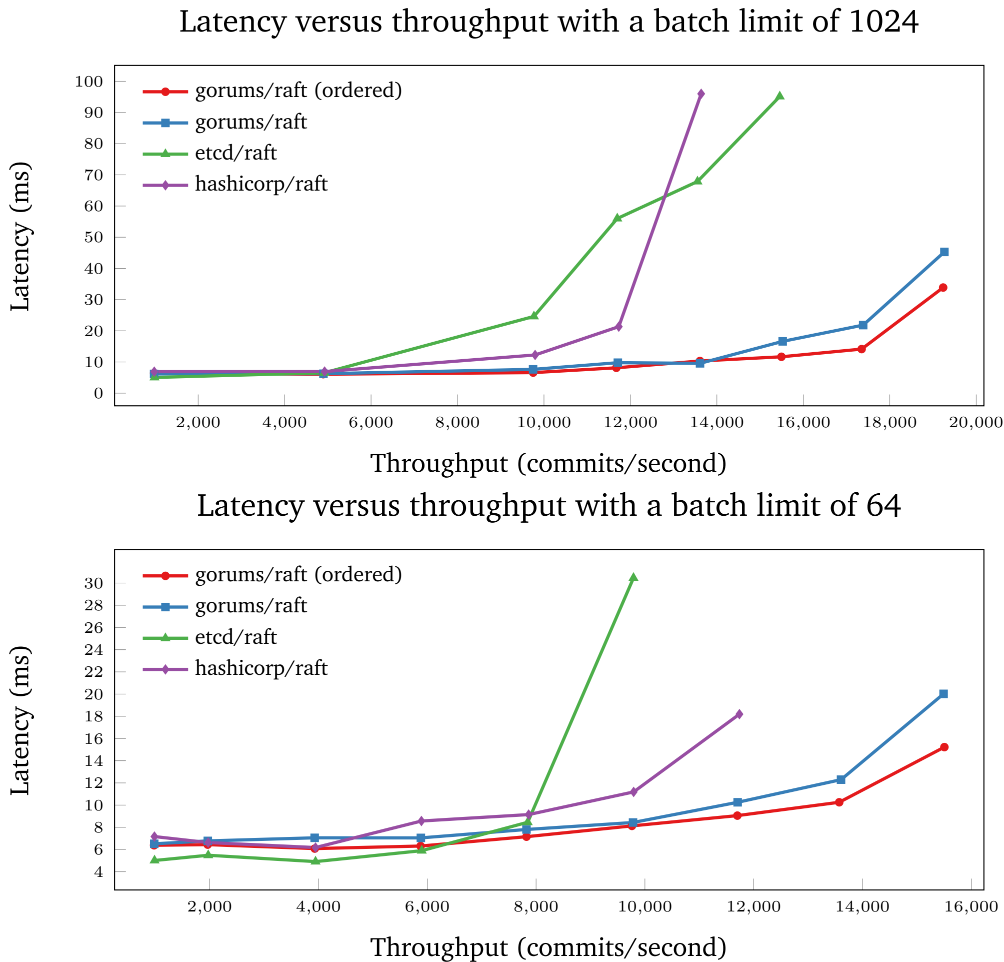


Figure 8.1: The latency at different client request loads with batch limit 1024 and 64, respectively. Each data point is the average of 4 experiments.

give us some insight into the overall performance of the implementations and how they compare. We should then be able to notice discrepancies in performance during further experiments.

We run this experiment using two different batch limits, 64 and 1024. As we use a payload size of 16 bytes per client request this equates to a maximum of 1 KiB and 16 KiB per AppendEntries. The latter is so high that the limit is never reached, so the latency is dominated by the heartbeat timeout which is set to 2 ms.

Figure 8.1 show the result of running this experiment 4 times for both batch limits for a variety of client requests per second for each Raft implementation. The measurements are done on the client. For gorums/raft we run the experiment with the Catchup extension and Strict Ordering op-

tion (order). Given a batch limit of 1024, we see a high throughput for all the implementations. However, hashicorp/raft and etcd/raft experience a rapid growth in latency as the throughput increases. We believe this is due to some AppendEntries RPCs failing given the high load, and the pipelining mechanism having to abort. The latency of gorums/raft grows steadily and only start to dip up after reaching the higher client request loads. We believe this is due to how we make use of quorum calls with Raft. When we invoke an AppendEntries quorum call, we are forcing all followers to decide on the provided entries at once. In traditional Raft implementations like hashicorp/raft and etcd/raft the replication of entries happens to each follower individually. This means that followers are not necessarily at the same index in the log.

With a batch limit of 64, we see much the same trend as with 1024. The latency and maximum throughput are however much lower. The batch limit is now low enough for it to be reached before a heartbeat timeout. This suggest that etcd/raft and hashicorp/raft should show a lower latency at moderate load than gorums/raft, which does not implement pipelining. The gorums/raft implementation will send one AppendEntries, then reach the batch limit again before receiving a response from a quorum. This causes some queuing and limits the minimum latency possible. We observe this difference at least between gorums/raft and etcd/raft, where the latter has better latency at moderate load. We expected to see this with hashicorp/raft as well, but it seems that we would have to lower the batch limit even more to observe the effect.

8.4 Network Partition

In this next experiment we study at the behavior of the different Raft implementations during a brief network partition. The experiment is designed to evaluate the Catchup extension to Raft and the Strict Ordering option for Gorums.

The experiment is similar to the common case operation experiment, except that we partition a single follower from the leader for 5 seconds. A fixed rate of 10000 client requests per second is issued. During the partition all messages between the leader and the partitioned follower are dropped. We are interested in examining how the leader brings the partitioned follower up to date with the rest of the cluster after the partition have been lifted.

Figure 8.2 shows the averaged result of running this experiment 10 times. The first graph describes the number of log entries committed per

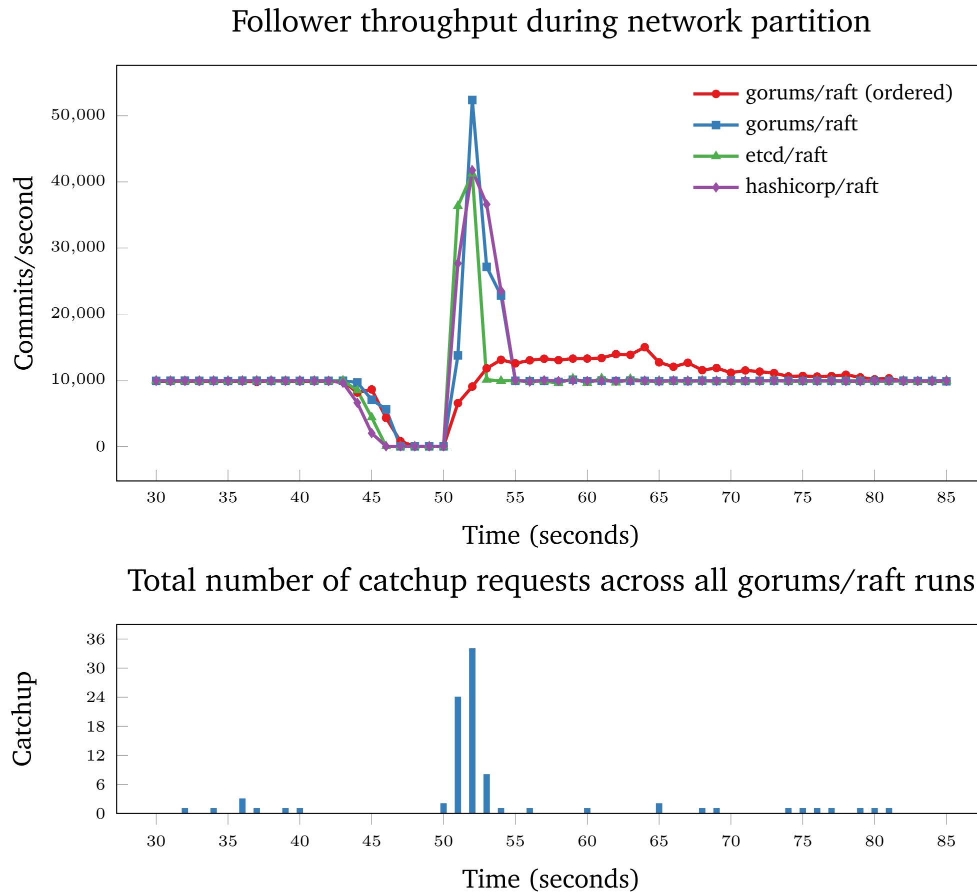


Figure 8.2: The first graph show the throughput (measured at follower) of different Raft implementations as a follower is experiencing a network partition. The partition takes place from around 44 to 49 seconds. The batch limit is 1024 and each data point is the average of 10 experiments. The lower graph depicts the total number of Catchup RPCs sent by the follower during all gorums/raft experiments.

second, as viewed from the partitioned follower. The number of entries committed, (the throughput) reveals how the follower recovers from the partition. The network partition appears in the graph between 44 and 49 seconds. Next, the bar chart shows the number of Catchup RPCs invoked by the partitioned follower during all runs of the experiment with the gorums/raft implementation.

We will first look at how hashicorp/raft and etcd/raft handles the partition. Both implementations show a sudden raise in throughput as the partition is lifted. Further investigation reveals that the implementations stop replicating entries when a connection is lost. This is reasonable as it is wasteful to send data when we are reasonably certain that it will not arrive

at the destination. When the connection is eventually restored, the leader continues where it left off. The partitioned follower is rapidly brought up to date.

Both etcd/raft and hashicorp/raft behave as dictated by the Raft algorithm in this regard. The method wastes no bandwidth sending superfluous log entries. In addition, the duration of the partition is irrelevant. The AppendEntries RPC to the individual follower is capable of negotiating the exact entries that the follower needs. The log replication to the follower can thus continue.

In blue, we see the throughput of our gorums/raft implementation with the Catchup extension enabled. The results look very similar to that of etcd/raft and hashicorp/raft. Looking at the bar chart, we see a surprising number of Catchup RPCs being invoked. Even when there is no partition, the follower issues Catchup RPCs. We were expecting to see a single Catchup RPC after the partition was healed.

The reason for this behavior have been thoroughly discussed in Section 7.3. It comes down to there being a chance AppendEntries RPCs are invoked out of order. This causes the follower to believe that it has lost one or more AppendEntries RPCs. The follower responds by sending a Catchup RPC request to the leader.

This experiment shows that the Catchup extension is able to prevent the situation that motivated its development, followers stalling on even a single missed log entry. The number of Catchup RPCs invoked are however discouraging. The leader is continuously retransmitting log entries that were eventually correctly received by the follower. Though the follower chooses to disregard the entries as they cannot be replicated safely. This behavior wastes resources that could be used elsewhere.

After discovering that the reordering of RPCs was due to how we applied the quorum call abstraction. We developed the Strict Ordering option for Gorums, and repeated the experiment with the option enabled. The result can be seen in red as gorums/raft (ordered). Note that the Catchup extension is still enabled.

With the reordering issue fixed, we expected to see exactly one Catchup RPC being invoked as discussed above. Again, to our surprise, no Catchup RPC is issued by the follower. In addition, the follower's throughput seems to recover in a somewhat strange pattern.

What we see is TCP retransmitting the packets that were lost due to the network partition. TCP guarantees that the packets are delivered in the same order they were sent. Thus, the Catchup RPC is never invoked as the follower does not receive any AppendEntries RPCs out of order. These results diminish the need for the Catchup RPC to some degree.

Table 8.3: Time to commit a client request during the network partition experiment (measured at client). Each data point is the average of 10 experiments.

Commit times (ms)				
Implementation	Mean	Stdev	Min	Max
gorums/raft (order)	7.85	7.41	2.79	230.55
gorums/raft	7.79	3.94	2.91	174.86
etcd/raft	29.84	16.09	1.90	88.68
hashicorp/raft	9.52	24.03	2.12	930.14

The reason this works is because the leader never learns that the follower is partitioned. This is again due the AppendEntries quorum call ignoring replies, as discussed in Section 7.2. As a result, the leader keeps sending AppendEntries RPCs to the follower. These RPCs are buffered by TCP, and when its buffer is full, our application will start to block on network operations. This hints at longer network partitions being a problem with our current implementation. The next experiment will describe a potential solution to this problem by using reconfiguration.

Table 8.3 shows the latency measured by the client during this experiment for each implementation. In spite of the problems we discovered with our implementation, gorums/raft manages to perform on par with hashicorp/raft and etcd/raft. An interesting observation is that hashicorp/raft have a much higher max latency. We find that this is due to a leader election occurring directly after the partition heals. Further investigation reveals that this is due to hashicorp/raft not implementing the pre-election phase, as both gorums/raft and etcd/raft do. If we did not implement the pre-election phase in gorums/raft, the follower would have rejected the retransmitted entries due the RPCs being from an older term. Thus, recovery through TCP retransmission would have been possible.

8.5 Reconfiguration Policy

We have shown that the Catchup extension can handle omission faults in the common case. Though, we also found out that lengthier network partitions may pose a problem with the Catchup approach. This problem applies only when less than a majority of the servers are partitioned or have crashed. The AppendEntries quorum call is able to recover from a rejected response, as long as a majority of the servers are operational. To handle situations where a minority of servers are experiencing issues, we propose a reconfiguration policy be implemented.

The idea is to have the leader monitor its followers for indications of failure. We introduced such a mechanism in Section 7.4. In addition, we could take a follower frequently issuing Catchup RPCs as a sign of a poor network connection, or the server having issues. With this information, the leader could perform a reconfiguration, replacing the faulty server, avoiding potential issues in the future.

One could take this approach one step further, and keep additional non-voting servers that receive log entries in the cluster. The log entries could be replicated from followers to reduce load on the leader. In fact, hashicorp/raft supports the leader replicating entries to non-voting servers that can later be promoted to servers with voting rights.

For the next experiment we have chosen to emulate such a reconfiguration policy. This allows us to evaluate the different reconfiguration implementations. In addition, we are able to examine the viability of our reconfiguration policy suggestion.

Our experiment will be conducted on a cluster of four servers. Initially only three of the servers are part of the cluster. We crash one server, causing the fourth server to be included through a reconfiguration to replace the crashed server. In fact, we chose to crash the leader of the initial cluster. This causes a new leader to be elected, which is then responsible for carrying out the removal of the crashed server. The reason for crashing the leader, and not a follower, is to allow us to observe the full reconfiguration process. Both gorums/raft and hashicorp/raft require a log entry to be committed in the leader's term before any configuration changes can take place.

We use a heartbeat of 2 ms, as with the other experiments. The leader is handling 1000 client requests per second during the experiment. The throughput is reduced as we are interested in demonstrating the difference in algorithm implementation and not performance. With a light load, it is easier to identify what the algorithms are spending time on.

Figure 8.3 shows this experiment for the different Raft implementation. Ticks on the vertical timeline identifies when different events occur. Thus, the time shown between two ticks, are the time between two events. The color of the background represents the current configuration. Note how different servers observe configuration changes at different times. The gorums/raft implementation has the Strict Ordering option enabled. The figures show a visualization of the process:

1. The leader S2 crashes and S1 is elected the new leader.
2. S1 detects that S2 have crashed, and that it has S4 available as a replacement.

3. S1 proposes to remove S2 from the cluster.
4. S2 is removed from the cluster.
5. S1 proposes to add S4 to the cluster.
6. S4 is added to the cluster.

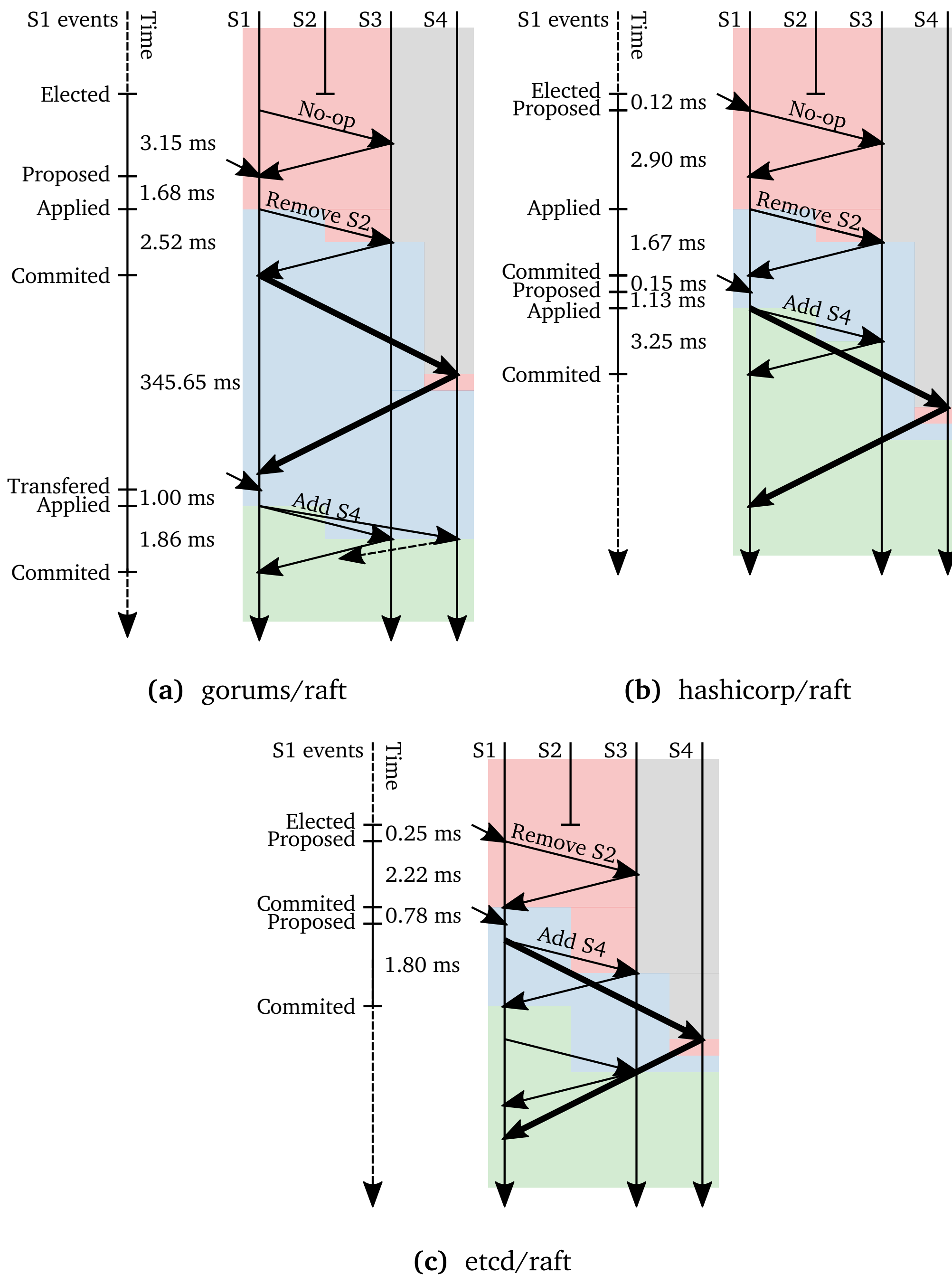
Whether or not it is better to add or remove the crashed server first is debatable and depends on the cluster size. In this experiment S2 is removed first for availability reasons. If we added S4 first, it would have been forced to participate in committing client requests. This is not be advisable as S4 stores no log entries on being included into the cluster. At least this is true for etcd/raft and hashicorp/raft, which do not catch-up new servers before reconfiguration. We will now discuss the details of the process for each implementation in turn.

8.5.1 gorums/raft

The experiment performed with the gorums/raft implementation can be seen in Figure 8.3a. Before the leader (S1) is allowed to perform any configuration changes, it must commit a log entry from its own term. We have discussed the reason for this in Section 2.5.3. Hence, we commit a *No-op* entry, which is ignored by the state machine. Immediately after the entry has been committed, S1 proposes to remove S2 from the cluster. This is followed by a pause as gorums/raft does not write entries to its log before an AppendEntries quorum call is due. The new configuration is activated and an AppendEntries quorum call executed. A response from S3 is received, committing the configuration. S1 can now start the process of adding S4 to the cluster. Before S1 actually proposes to add S4 to the cluster, it initiates a parallel process replicating log entries to S4 as indicated by the thick arrows. When S4 stores enough entries to participate in committing new client requests, S1 proposes the configuration to include S4. This event coincides with the *Transfered* event in the figure. On the next AppendEntries quorum call, S1 appends the configuration change entry to its log. A response is received first from either S3 or S4, committing the configuration. Thus, gorums/raft implements reconfiguration exactly as described by the single-server membership change algorithm.

8.5.2 hashicorp/raft

The hashicorp/raft process in Figure 8.3b is similar to that of gorums/raft. The first difference is that hashicorp/raft allows configuration changes to



(a) gorums/raft

(b) hashicorp/raft

(c) etcd/raft

Figure 8.3: The leader (S2) crashes and S1 is elected as the new leader. S1 detects that S2 is not responding and that it has a backup server (S4) available. It then proposes to exclude S1 from the cluster reducing the quorum size to 2, making it equal the cluster size. After the new configuration is committed, another reconfiguration process is initiated to add S4, thus bringing the cluster size back to 3. Figure a, b and c show the process for the different Raft implementations.

be proposed before the leader has committed an entry in its term. This works because the configuration is not added to the log until an entry is committed. Next, hashicorp/raft deviates from the single-server membership algorithm by immediately adding S4 as a voting member. As we have discussed earlier in Section 2.5.3, this makes hashicorp/raft vulnerable to availability loss. In addition, we cannot reliably measure when S4 is actually caught up with the rest of the cluster. Hence, the state transfer time not being visible in the figure.

8.5.3 etcd/raft

Both gorums/raft and hashicorp/raft applies a proposed configuration on adding it to their log. In etcd/raft, which can be seen in Figure 8.3c, an alternate approach is implemented. Configuration changes only take effect after being committed. Thus, committing a log entry on becoming leader is not required for reconfiguration. As with hashicorp/raft, etcd/raft does not implement a catch up phase before adding servers to the cluster. Thus, etcd/raft is also vulnerable to availability loss when adding new servers to the cluster.

9 Conclusion and Further Work

This chapter concludes this thesis and presents suggestions for further work.

9.1 Conclusion

Gorums is built to aid in the design and implementation of quorum based systems. In this thesis, we have attempted to verify if this holds for the implementation of the Raft consensus algorithm. Through building a practical replicated service that support reconfiguration, we make our findings.

We show that a Raft implementation can be created with Gorums, that perform on par with the state of the art implementations. This is demonstrated through experimentation. We do however find that Gorums' quorum call abstraction does not play well as a replacement for Raft's AppendEntries RPC. Constructing a reliable log replication process using a quorum call was a difficult task. We take this as evidence that Gorums does not simplify the implementation of log replication in Raft, quite the opposite.

The function of Raft's AppendEntries RPC is quite dense. It distributes log entries, maintains a heartbeat, handles commitment, and ensures that followers recover log entries after failures. Consequently, we believe that greater success will be had in implementing consensus algorithms with better separation of concerns.

To contrast, we experienced no difficulties in the implementation of reconfiguration with Gorums. As quorum calls are bound to the configuration object, reconfiguration is a simple matter of creating a new configuration

and replacing the active configuration object. The next AppendEntries quorum call is simply invoked on the new configuration.

We acknowledge that the simplicity of the single-server membership change algorithm should be attributed in part for the ease of implementation. We also find reconfiguration in etcd/raft and hashicorp/raft to be quite straightforward. The etcd/raft and hashicorp/raft implementations does however not implement a catch up phase prior to adding servers to the cluster. As a consequence, they are especially vulnerable to unavailability during reconfiguration. This indicates that the implementation of the catch up phase is non-trivial. With Gorums the implementation of the catch up phase was straightforward. We credit this to the use of the quorum call abstraction.

We believe that Gorums provides well thought out abstractions that simplify many aspects of building distributed systems. While the Gorums framework is fairly small and easy to learn, it takes time to gain the insights needed to best apply the abstractions in practical applications. We do however predict that this will become easier when more systems are developed with Gorums, as more examples, common idioms, and best practices have had time to form.

9.2 Further Work

This section provides some thoughts on further directions from the work presented in this thesis.

Arbitrary Configuration Changes The single-server membership algorithm is simple. It would be interesting to implement a more complex algorithm that supports arbitrary configuration changes. In doing so, we believe one can show more prominently that Gorums simplify reconfiguration.

Paxos Implementation The problems we encounter with log replication is mainly caused by Raft not allowing gaps in followers' logs. Paxos does not have this restriction. Hence, a study on the Paxos protocol might unveil more appropriate uses for the quorum call abstraction.

Divergence Detection A quorum call provides a simple way to communicate with a quorum of servers. However, this simplicity comes at a price as it becomes harder to reason about the individual server's state. An event-based system could be built to detect when individual servers act different than the quorum, e.g., by responding differently

or simply by crashing, thus triggering an event and allowing appropriate actions to be taken.

Model-Based Testing We briefly mention model-based testing in Section 8.1, it would be interesting to examine if generating tests for our quorum calls from such a model could have revealed the issue with reordered RPCs earlier.

A
Experimental Data

This appendix contains the complete experimental data for the common case operation experiment in Section 8.3.

Table A.1: Common case operation of gorums/raft.

Batch limit: 1024			
Throughput (commits/s)		Latency (ms)	
Mean	Stdev	Mean	Stdev
981.16	6.71	6.40	1.16
4893.32	28.34	6.08	2.74
9750.23	41.41	6.59	1.52
11674.64	73.26	8.13	1.96
13608.85	161.63	10.31	6.19
15493.78	131.19	11.67	3.82
17349.42	189.75	14.13	3.88
19235.56	317.53	33.87	13.81

Batch limit: 64			
Throughput (commits/s)		Latency (ms)	
Mean	Stdev	Mean	Stdev
978.69	14.34	6.38	1.81
1963.58	14.27	6.44	1.34
3932.47	20.38	6.08	1.28
5877.10	25.10	6.31	1.38
7823.26	34.23	7.16	1.57
9761.62	54.52	8.13	3.62
11699.51	65.02	9.05	2.49
13571.71	104.60	10.26	3.29
15504.84	149.00	15.22	5.76

Table A.2: Common case operation of gorums/raft (order).

Batch limit: 1024			
Throughput (commits/s)		Latency (ms)	
Mean	Stdev	Mean	Stdev
980.17	8.91	6.18	2.38
4898.52	25.02	6.24	2.38
9747.97	57.43	7.63	1.93
11709.09	85.37	9.77	5.35
13610.63	95.19	9.56	5.62
15522.56	225.33	16.61	7.41
17385.00	291.11	21.83	12.94
19264.19	353.77	45.29	28.09

Batch limit: 64			
Throughput (commits/s)		Latency (ms)	
Mean	Stdev	Mean	Stdev
981.65	12.62	6.52	3.25
1965.77	12.86	6.78	2.65
3930.05	24.79	7.05	3.35
5882.06	33.27	7.05	2.34
7823.10	39.08	7.80	4.93
9782.02	63.53	8.43	3.17
11704.65	85.25	10.25	4.44
13604.82	126.81	12.29	6.89
15489.89	327.48	20.02	18.08

Table A.3: Common case operation of etcd/raft.

Batch limit: 1024			
Throughput (commits/s)		Latency (ms)	
Mean	Stdev	Mean	Stdev
988.63	5.61	5.04	2.63
4915.73	28.30	6.53	3.85
9767.72	108.44	24.61	10.99
11696.10	148.53	56.00	9.40
13551.69	262.18	67.86	11.54
15455.30	387.57	95.12	17.76

Batch limit: 64			
Throughput (commits/s)		Latency (ms)	
Mean	Stdev	Mean	Stdev
987.01	8.57	5.01	2.31
1973.92	11.94	5.48	3.79
3946.22	23.39	4.91	2.10
5898.69	27.43	5.90	2.90
7846.21	52.79	8.46	4.48
9790.32	144.45	30.46	12.34

Table A.4: Common case operation of hashicorp/raft.

Batch limit: 1024			
Throughput (commits/s)		Latency (ms)	
Mean	Stdev	Mean	Stdev
984.40	11.79	6.89	3.00
4925.51	26.64	6.93	2.15
9795.59	88.00	12.25	7.48
11730.80	125.63	21.33	9.11
13637.40	338.60	95.99	36.41

Batch limit: 64			
Throughput (commits/s)		Latency (ms)	
Mean	Stdev	Mean	Stdev
985.24	27.34	7.17	2.96
1972.75	12.42	6.63	2.94
3945.98	26.59	6.18	1.83
5892.52	51.02	8.57	4.36
7863.06	93.62	9.15	4.94
9793.18	67.76	11.19	3.64
11737.67	101.88	18.19	6.81

B

Attachments

This appendix contains the full code work for this thesis, as well as links to the version controlled repositories.

- The implementation of the Raft consensus algorithm with Gorums.
  - Embedded: raft.7z
  - Repository: `https://github.com/relab/raft`
- The implementation of the Raft Key-Value Store, Ansible playbooks and utility programs developed.
  - Embedded: rkv.7z
  - Repository: `https://github.com/relab/rkv`
- Printer friendly version of this thesis with additional blank pages, no formal front page and no attachments appendix.
  - Embedded: Pedersen_Sebastian-print.pdf

Bibliography

[1] M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 336–345, June 2005.

[2] Amazon.com, Inc. Amazon EC2 Instance Types.
`https://aws.amazon.com/ec2/instance-types`.

[3] Amazon.com, Inc. Amazon Elastic Compute Cloud (Amazon EC2).
`https://aws.amazon.com/ec2`.

[4] James W. Anderson, Hein Meling, Alexander Rasmussen, Amin Vahdat, and Keith Marzullo. Local recovery for high availability in strongly consistent cloud services. *IEEE Trans. Dependable Sec. Comput.*, 14(2):172–184, 2017.

[5] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[6] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Berlin Heidelberg, 2011.

[7] CoreOS, Inc. etcd.
`https://coreos.com/etcd`.

[8] CoreOS, Inc. etcd's raft (source code).
`https://github.com/coreos/etcd/tree/master/raft`.

[9] CoreOS, Inc. Website.
`https://coreos.com`.

[10] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[11] Google Inc. gRPC.
`http://www.grpc.io`.

[12] Google Inc. Protocol Buffers.
`https://developers.google.com/protocol-buffers`.

[13] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 169–184, Berkeley, CA, USA, 2016. USENIX Association.

[14] HashiCorp. Consul.
`https://www.consul.io`.

[15] HashiCorp. raft (source code).
`https://github.com/hashicorp/raft`.

[16] HashiCorp. Website.
`https://www.hashicorp.com`.

[17] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: Do we have consensus? *SIGOPS Oper. Syst. Rev.*, 49(1):12–21, January 2015.

[18] Leander Jehl, Tormod Erevik Lea, and Hein Meling. Replacement: Decentralized failure handling for replicated state machines. In *34th IEEE Symposium on Reliable Distributed Systems, SRDS 2015, Montreal, QC, Canada, September 28 - October 1, 2015*, pages 156–165, 2015.

[19] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 245–256, Washington, DC, USA, 2011. IEEE Computer Society.

[20] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[21] Leslie Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, December 2001.

[22] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.

[23] Tormod Erevik Lea, Leander Jehl, and Hein Meling. Gorums: New Abstractions for Implementing Quorum-based Systems. Unpublished (Extended Version), 2016.

[24] Tormod Erevik Lea, Leander Jehl, and Hein Meling. Towards New Abstractions for Implementing Quorum-based Systems. In *37th IEEE Intl. Conf. on Distributed Computing Systems (ICDCS)*, 2017.

[25] Tormod Erevik Lea, Leander Jehl, Hein Meling, and Sebastian Mæland Pedersen. Gorums (source code).
`https://github.com/relab/gorums`.

[26] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, 2012.

[27] Hein Meling, Alberto Montresor, Bjarne E. Helvik, and Özalp Babaoglu. Jgroup/ARM: a distributed object group platform with autonomous replication management. *Softw., Pract. Exper.*, 38(9):885–923, 2008.

[28] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[29] Diego Ongaro. Bug in single-server membership changes.
`https://groups.google.com/forum/#!topic/raft-dev/t4xj6dJTP6E`.

[30] Diego Ongaro. LogCabin.
`https://github.com/logcabin/logcabin`.

[31] Diego Ongaro. Raft consensus algorithm implementations.
`https://raft.github.io/#implementations`.

[32] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014.

[33] 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, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[34] Sebastian Mæland Pedersen. Raft Key-Value Store (source code).
`https://github.com/relab/rkv`.

[35] Sebastian Mæland Pedersen. Raft with Gorums (source code).
`https://github.com/relab/raft`.

[36] Sebastian Mæland Pedersen. Raft Consensus with Gorums. Technical report, University of Stavanger, 2016.
`https://github.com/relab/raft/files/1073677/raft.pdf`.

[37] Red Hat, Inc. Ansible.
`https://www.ansible.com`.

[38] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[39] Nuno Santos and André Schiper. Optimizing paxos with batching and pipelining. *Theor. Comput. Sci.*, 496:170–183, July 2013.

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

[41] The Go Authors. The Go Programming Language.
`https://golang.org`.

[42] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.

[43] Rui Wang, Lars Michael Kristensen, Volker Stolz, and Hein Meling. Application of Model-based Testing on a Quorum-based Distributed Storage. In *International Workshop on Petri Nets and Software Engineering - PNSE'17*, 2017.