Raft Consensus with Gorums

Sebastian M. Pedersen
sem.pedersen@stud.uis.no
University of Stavanger

Abstract

In this project we implement the Raft Consensus Algorithm using Gorums. Several algorithms have been implemented with Gorums, this is the first time Raft is implemented. As both Raft and Gorums are designed to help system builders, we study the benefit of using them together. We focus on the evaluation of Gorums as a tool to implement a distributed system, and how well Raft matches with the data-centric model used by Gorums. The impact Gorums have on performance is also taken into account. Our results show that Gorums is a well thought out tool, a big help in the building distributed systems. We do however experience that Raft and Gorums do not go well together. We conclude that this is rooted in Raft breaking with the single responsibility principle.

1 Introduction

It is well known that the building of fault tolerant distributed systems is hard to get right. In this project, we will explore two works which aims to make this easier. The first work is Raft [11], a consensus algorithm designed with understandability in mind. The other is Gorums [15], a framework made to simplify the design and implementation of quorum-based systems. We put both of these works to the test by building an implementation of Raft with the Gorums framework. We are interested in seeing how well Raft implements with Gorums. Our focus will be on how the abstractions presented in Gorums helps with the implementation of a distributed consensus algorithm, such as Raft. We will also look at how the use of Gorums impact performance. Prior to this project we have implemented Paxos [7, 8], without the use of any third-party libraries. We have thus experienced the difficulty of implementing a distributed consensus algorithm firsthand. Building on our experience, this project will allows us to discover if Raft and Gorums combined, can help us reduce some of the effort.

Here are the key points of our project:

- We built an implementation of the Raft Consensus Algorithm using Gorums.
- We evaluate our implementation in three different configurations in terms of throughput and latency. One configuration making only limited use of Gorums, while the others make trade-offs between latency and performance.
- We use our experience in Gorums, gained building Raft, to discuss what we think are the benefits and challenges of using Gorums.
- We discuss the problems experienced with Raft, and criticize Raft's disregard of the single responsibility principle.
- We propose an extension to Gorums easing the implementation of Raft, and potentially other algorithms.
- We conclude with reporting on how well Gorums have performed in the task of implementing Raft.

2 Background

In this section we will give an informal description of Gorums and Raft. We will present the key concepts that we think are relevant to this project.

2.1 Gorums

Gorums [15] is a new framework which serves to simplify the construction of fault tolerant distributed systems. The framework provides an abstraction for transparently invoking a *remote procedure call* (RPC) on a group of servers. This abstraction is called a *quorum call*. The replies are collected and combined into a single

reply. A user-defined *quorum function* is invoked on the collection of replies each time a new reply is received. The function is responsible for combining the replies, and indicating to the quorum call when enough replies have been collected. Usually enough replies have been collected when the quorum function can determine if the process will *abort* or *commit*. The quorum call will then return to the caller with a single reply, relieving the user of the coordination required to communicated with multiple servers. Gorums acknowledges the need to invoke RPCs on different groupings of servers, and therefore allows for the creation of different *configurations* on which quorum calls can be invoked. Configurations are a low-cost abstraction, local to the caller, which can be created on-demand. Quorum calls are exposed as methods on these configuration objects.

The Gorums paper [15] presents several case studies which evaluate the usefulness and performance of the framework. Gorums builds on Google's gRPC [2] and Protocol Buffers (protobuf) [3] libraries.

2.2 Raft

Raft [11] is a fairly new algorithm for consensus in a distributed system. Raft was designed specifically to address the problem of Paxos [7, 8] being notoriously difficult to understand, leading to a divergence in interpretations and implementations. The consensus algorithm is leader-based similar to ViewStamped Replication (VR) [10, 9] and ZooKeeper Atomic Broadcast (Zab) [6]. Raft greatly reduces the number of different messages types and RPCs that must be implemented compared to these two. Raft is used to manage a replicated log, which usually powers a *replicated state machine* (RSM). There exists a multitude of Raft implementations [13] in various stages of development. And there has been an attempt [5] to evaluate the claims about Raft made by its authors.

Raft is a leader-based algorithm, and as such all client requests go through the leader, and other servers do rarely interact directly with each other. A server can be leader in a given term, no two servers can be leader in the same term. This is Raft's election safety guarantee. A Raft server can be in one of three states: *follower*, *candidate* or *leader*. A server always begins in the follower state. If the server receives no requests from a server in the leader or candidate state, the server will transition into the candidate state after some random timeout. In the candidate state, the server will request votes from other servers, becoming the leader if it can gather a vote from the majority. A candidate always vote for itself. This phase is driven by the *RequestVote* RPC, which the candidate invokes on all the other servers. If a response is received, the candidate stops invoking the RequestVote RPC for that particular server. Eventually the server will do one of three things: receive enough votes and become leader, see a higher term and become a follower, or reach a timeout and start an election with a higher term.

After a leader is elected, Raft is able to make progress managing its replicated log. Log replication in Raft is completely handled by the *AppendEntries* RPC. This RPC has several responsibilities, mainly to distribute new log entries but also serves as a heartbeat to prevent followers from timing out. Another function is informing the leader of the followers state. A follower will successfully replicate the received entries only if its term and log matches that of the request. The success of the RPC is important as it tells the current leader which servers are missing log entries, or if it needs to step down to the follower state in the case of a response containing a higher term. A server must always update its term and step down to the follower state when discovering a higher term. If the AppendEntries RPC fail for a given server because the log mismatched, the leader will move one step back in the log and retry. As this can potentially happen a large number of times, for a server that is far behind, we implement an optimization. Followers will respond with a *MatchIndex* for the exact entry in the log that the follower needs next.

3 Design & Implementation

Our implementation of Raft is written in Go [14], as Gorums is a Go library. The source code is freely available [12].

The basic Raft Consensus Algorithm operates using only two RPCs, namely the RequestVote and AppendEntries RPC. The RequestVote RPC handles leader election, and AppendEntries is used to replicate log entries to followers. The AppendEntries RPC also functions as a heartbeat, preventing followers from starting new elections. Using the Gorums framework, the core of our implementation will be in designing quorum calls which implement the behavior of the RequestVote and AppendEntries RPC. We will now discuss some of the challenges encountered while implementing these quorum calls, and then present our actual implementations.

3.1 Challenges

During the development of the RequestVote and AppendEntries quorum calls, we were faced with several challenges. We will first present two minor problems that were encountered when converting both RPCs to quorum functions. We then discuss some more serious challenges with the AppendEntries implementation.

3.2 Aborting a Quorum call

The Raft algorithm specifies that a server which encounters a higher term than its current term, should immediately update its term and revert to follower state.

This is problematic to implement in Gorums as a quorum function does not have access to the Raft server's internal state. There are ways to allow quorum functions access, but after consulting with the Gorums authors, we have yet to find an idiomatic way to do this in Gorums.

As knowing the term in which a RPC was initiated is beneficial, we augment the RPCs to include the request term. This changes the RPCs to better fit the Gorums model while retaining its initial properties.

This change incurs a small bandwidth cost for information that is already present on the server. In this case, it is not a huge price to pay, but we believe this problem can be avoided entirely. We propose a solution to this problem in Sect. 5.2.

3.3 Canceled Replies

After issuing a RPC, the Raft algorithm can proceed after receiving a majority of the replies. The remaining replies will be processed as they are received. As we have earlier established, if any of these replies contain a higher term than the server, the server must convert to the follower state.

When a quorum function returns after aborting or receiving a quorum of replies, the quorum call will cancel all outstanding replies. As such, the server will not process any of these replies. If any of the canceled replies contained information that the server could have acted upon, that information is lost.

Thankfully this does not affect the correctness of the algorithm. The canceled replies are no different than missing replies, which the algorithm is designed to tolerate. Worst-case scenario is a server discovering a new term a little late, which is not a problem. If we assume elections are far between, this is not something that will occur very often.

Heartbeat One important observation here is that, followers receive the issued requests, it is their replies which is canceled. This means that the heartbeat functionality works as expected. It is only the leader which will chose to not process the remaining replies in the case a quorum function returns early.

3.4 Separation of Concerns

The AppendEntries RPC has a collection of responsibilities. In our implementation there are three important ones: replicating log entries, the heartbeat mechanism, and informing the leader which entries to send to each follower next. When replicating log entries, a majority of successful replies suffice to commit the entries. For performance reasons it makes sense not to wait for the remaining replies. However each individual response contains important information about a follower. This information is lost, if we do not wait for all the replies.

In Gorums a quorum call can only return once, generating a single reply. Now if we want an optimal AppendEntries implementation for replicating log entries, the quorum function should return as soon as it has a majority of successful replies. As we discussed above, when a quorum function returns the quorum call will not process the remaining replies. This is of course problematic as the leader does not know if the missing replies were successful, or what entries they were requesting. We could force the quorum function to process all replies. However if the quorum call needs to wait for replies from every server, and not just a majority quorum, the latency of the request will be reduced to that of the slowest server. We would rather the latency be contacting the fastest quorum, which is a key property of the Raft algorithm.

We do not present an optimal solution to this problem, as it would require decomposing the AppendEntries RPC into individual RPCs. This creates the problem of retaining the correctness of the algorithm. We have chosen to implement a quorum call that can reach both a majority and the full cluster, depending on the quorum size is set.

Optimal replication The performance of Raft is dependent on how fast the AppendEntries RPC can be invoked. Our first quorum call therefore uses a quorum size equal to the majority of the cluster. This allows the quorum call to commit as soon as a majority of the servers have responded successfully.

The problem with this approach stems from the canceled replies, as we discussed in the previous section. As only a majority of the replies are processed, we have to be optimistic and assume that the remaining followers were successful as well.

In practice we have experienced that this is not true. Even in an ideal *local area network* (LAN) setting, followers tend to eventually fall behind. The leader is never informed of the log entries needed by the slower servers. This causes a follower who fall behind, to stay behind. Only after something happens to one of the faster servers, or a new leader is elected, will the other followers have a chance to receive the missing entries.

If we assume our cluster to be fairly stable, the cost of updating a slower follower, when the composition of faster followers change, can be dramatic. The longer a cluster operates without problems, the higher the cost of updating a follower will be. The algorithm can not make any progress until there is a majority of followers with

up-to-date logs.

A possible solution to this problem is to replicate entries directly to the slower servers which fall behind. We would then need a traditional AppendEntries RPC, capable of updating followers individually. However if we do implement this optimization, we have done the same work required not using a quorum call at all. A better way would be to implement *snapshotting*, which is a form of log compaction discussed in the Raft paper. We would then implement the additional *InstallSnapshot* RPC [11] required, and use it to periodically synchronize the replicated state machine. This would stop the gap in entries replicated to expand infinitely. However there is a cost in doing log compaction too regularly.

Synchronized cluster As there are obvious drawbacks with the previously discussed quorum call, we try increasing the quorum size to the full cluster size. This should make the quorum function wait until a reply from every follower has been received. With the information of the full cluster, we are able to construct new quorum calls containing the correct number of log entries to satisfy all followers.

The immediate problem with this quorum call is that latency will be determined by the slowest follower. Another issue is when servers stop responding. The leader will be unable to proceed until the quorum call times out. As we can not tolerate the cluster becoming unavailable when a single server crashes, this problem must be resolved.

We solve this problem by keeping a short timeout on the quorum call. We allow the quorum function to return successfully without a response from every server. Since the quorum function is designed to wait for all replies, success in the case of less replies, can only happen when the quorum call times out. This is why the timeout must be short. As such our quorum function should, in the common case, be able to successfully update every follower in time. If not every server respond in time, we can still proceed given that at least a majority responded successfully.

Another problem with this quorum call, is the fact that when a follower falls behind, every other server will receive an AppendEntries RPC with excess log entries. This causes additional bandwidth overhead. We believe this case to be rare, as the quorum call is specifically designed to make sure every follower stays up-to-date.

3.5 Combining Replies

In the previous section we describe a quorum call capable of making progress with a majority quorum. For this to work, we need to be able to figure out which followers were in that quorum.

Algorithm 1 Raft RequestVote quorum function

```
 1: func (qs QUORUMSPEC) RequestVoteQF(replies []RESPONSE)
 2:     votes := 0                                        ▷ votes granted
 3:     reply := replies[len(replies) − 1]                ▷ latest reply
 4:     if reply.Term > reply.RequestTerm then
 5:         return reply, true              ▷ abort if follower in higher term
 6:     for r := range replies do                         ▷ count votes
 7:         if r.VoteGranted then
 8:             votes++
 9:     if votes ≥ qs.FastQSize then
10:         reply.VoteGranted := true
11:         return reply, true                            ▷ quorum found
12:     return nil, false          ▷ no quorum yet, await more replies
```

This is not practical to implement in Gorums. While Gorums allows us to inspect which servers responded to a quorum call, we do not know which of them were successful. The traditional AppendEntries RPC contains a unique identifier for the follower responding. This allows the caller to update the corresponding MatchIndex. This is of course also possible in Gorums, however we are limited to setting one identifier, as all the replies are combined into a single reply.

We implement a simple solution to this problem. By allowing the replies to contain a list of ids instead of a single one, the quorum call can return a list of the followers who were successful.

As sending a list incurs more bandwidth, augmenting the RPC is not without cost. AppendEntries is the most used RPC in Raft, so this change is a lot more noticeable than the change to the RequestVote RPC in Sect. 3.2.

The problem of type discrepancy is described in the Gorums paper [15]. There is an ongoing discussion about allowing quorum calls to specify a different return type for quorum functions. Different from that of the replies. This would solve the problem with the bandwidth cost, but it remains to be seen if this change will be implemented.

3.6 Leader Election

Overall the implementation of the RequestVote quorum call was straightforward.

Algorithm 1 presents the RequestVote quorum function developed. First we check whether the latest reply contains a higher term, in which case we abort (L4). We iterate over the replies counting the number of votes granted (L6). If a majority granted a vote, we have received a quorum and can return (L9). In the case of no quorum and there are still remaining replies, we wait for more replies (L12).

Algorithm 2 Raft AppendEntries quorum function

```
1:  func (qs QuorumSpec) AppendEntriesQF(replies []Response)
2:      successful := 0                                  ▷ successful replies
3:      maxIndex := 0                                    ▷ max match index
4:      reply := replies[len(replies) − 1]               ▷ latest reply
5:      reply.Success := false
6:      reply.FollowerID := nil
7:      if reply.Term > reply.RequestTerm then
8:          return reply, true                           ▷ abort if follower in higher term
9:      for r := range replies do
10:         if r.MatchIndex < reply.MatchIndex then
11:             reply.MatchIndex := r.MatchIndex         ▷ lower match index
12:         if r.Success then                            ▷ count successful replies
13:             successful++
14:             maxIndex := reply.MatchIndex
15:             reply.FollowerID := reply.FollowerID ∪ r.FollowerID
16:     if successful = qs.SlowQSize then
17:         reply.MatchIndex := maxIndex
18:         reply.Success := true
19:         return reply, true                           ▷ slow quorum found
20:     if len(replies) = successful then                ▷ all replies must be successful
21:         if len(replies) ≥ qs.FastQSize then
22:             reply.Success := true                    ▷ majority successful
23:     return reply, false                              ▷ no quorum yet, await more replies
```

3.7 Log Replication

Developing a satisfactory quorum function for the AppendEntries RPC proved to be quite difficult. In Gorums it is possible to bypass the quorum function, so we have made an implementation using a regular RPC for AppendEntries. We used this RPC as baseline. Further we have two implementations using the same quorum function, the difference is in how we have set the quorum sizes. One is tuned for keeping every server up-to-date, and the other is tuned for speed, always trying to choose the fastest way to make progress. This might leave some servers with an outdated log. The difference in configuration in the first case, is using a quorum size equal to the cluster size, while the second case sets the quorum size to the size of a majority. In addition the first case is optimized to allow progress if some servers are down, as long as there is still a majority. Though this requires waiting for the request to time out.

Algorithm 2 shows the AppendEntries quorum function created. The value of *SlowQSize* and *FastQSize* dictate if the quorum function will update all or only the fastest quorum. If we set both *SlowQSize* and *FastQSize* to a majority of the cluster, the algorithm will try to operate on a fast quorum. Keeping *FastQSize* as the majority and changing *SlowQSize* to the size of the cluster, causes the quorum function to try to update all the servers. In the original AppendEntries RPC description, *FollowerID* contains the unique id of a single server. In our implementation we had to change it to a list. This is so that we could retain who successfully responded to the quorum call, when combining replies.

AppendEntries will abort, like RequestVote, in the case of a follower being in a higher term (L7). In the case of a slow quorum we need to find the minimum MatchIndex (L10). This is to make sure that the next quorum call is capable of successfully updating ever follower. Servers that are already up-to-date will ignore the excess log entries received. The followers which were successful are counted and saved (L12). We save the id of successful followers so that we know which log entries to send to these servers next. If we have enough successful replies to form a slow quorum, we can successfully return to the caller (L16). One important thing to note here is that, if *SlowQSize* is set to a majority, this is actually a majority quorum. The remaining code ensures liveness in the case $SlowQSize > FastQSize$ and less than a majority of the servers stop responding (L20).

4 Evaluation

Our evaluation focuses on quantifying the throughput and latency of the different AppendEntries RPC implementations. These results are meant to highlight some problems encountered during the design of our implementation. We note that we try to show the difference in the methods presented, and that the results are not necessarily good for absolute comparison with other systems.

We first describe the environment in which the evaluation were performed. Three AppendEntries RPC implementations are then presented, and evaluated regarding to throughput and latency.

4.1 Experimental Setup

All experiments were run on clusters of 512MB droplets (virtual machines) with a one core processor from DigitalOcean[1] running Ubuntu 16.10. All code is compiled using Go 1.6.2. Clusters of three servers were created, two servers in the region LON1 (London) and one in NYC3 (New York). The leader of the cluster is always in LON1. For measuring throughput the client is placed in the same region as the leader (LON1). As the client for measuring latency requires a lot of memory, the client is run locally. We call this region NOR (Norway). The latency from NOR to LON1 is 40ms, and from LON1 to NYC3 71ms. The reason for placing one server in New York is so that we can see the effect of one slow server in the cluster. All experiments are run over a period of 30 seconds, with an initial 15 seconds to saturate the cluster. The interval between each AppendEntries is set to 250ms giving a margin on the election timeout with NYC3 being 71ms away, while being the timeout that gave best results measuring throughput.

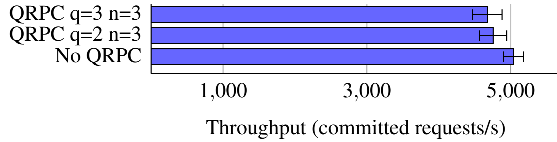

Figure 1: Throughput w/ standard deviation using batching on a cluster of 3 servers. 15 clients continuously trying to commit a request of 16 B. Clients are asynchronous. Average over 5 runs.

4.2 AppendEntries RPC Implementations

We will now present the three AppendEntries RPC implementations used during evaluation. We have already discussed this in Sect. 3.7, but we will name the implementations and highlight the differences here.

QRPC is shorthand for quorum call. We use q to indicate quorum size, and n cluster size, meaning $q \leq n$. In all cases the cluster size is 3.

No QRPC Does not use Gorums' quorum call functionality. Close to regular gRPC but retains the benefit of Gorums' manager. This implementation is most similar to the original Raft description. *No QRPC* has the configuration *q=2 n=3*.

QRPC q=2 n=3 Uses a quorum call with cluster majority as the quorum size. This configuration trades commit latency for slower servers falling behind. In this case, the NYC3 server is expected to end up with an outdated log, while the two LON1 servers proceed.

QRPC q=3 n=3 Uses a quorum call with the cluster size set as the quorum size. This configuration keeps every server up-to-date at the cost of higher commit latency. In this case, the NYC3 server is expected to be a bottleneck, causing the overall commit latency to increase. As to not disrupt the algorithm when a server crashes, this implementation allows committing an entry as long as at least a majority quorum succeeded. The logic can be seen in the last lines of Algorithm 2. This does however only work as a request times out and returns to the caller. This is because the quorum function is designed to keep waiting on replies from the whole cluster. We therefore expect to see a latency greater than the quorum call's timeout, when the cluster is not operating at full capacity.

4.3 Throughput

The objective of this experiment is to determine the effect that implementation choice has on the overall throughput of the cluster.

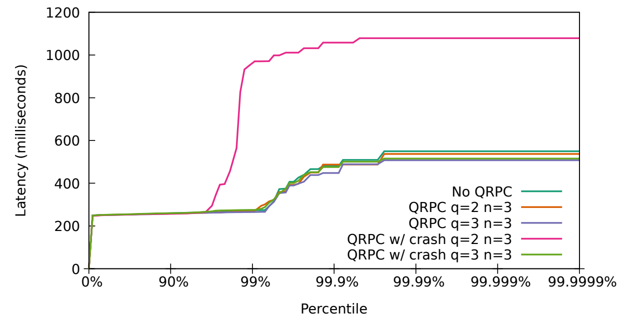

Figure 2: Latency using batching on a cluster of 3 servers. 15 clients trying to commit a request of 16 B. Clients are synchronous. Average over 5 runs.

During this experiment we enable batching. Batching trades latency for an increase in throughput. As the AppendEntries RPC is used for replicating entries to the cluster, batching is implemented by buffering client requests until the next AppendEntries RPC, which is sent every 250ms. There is no batching of client requests on the client side. As there is no application resting on top of our Raft implementation, throughput is entirely measured in client requests committed to the replicated log.

Figure 1 shows the throughput for the different implementations using a request size of 16 B, with 15 asynchronous clients. We did not gain any additional throughput by increasing the number of clients past 15. *No QRPC* has a slightly higher throughput than the *QRPC* implementations. Our key take away from this experiment is that throughput is not greatly affected when using a quorum call.

4.4 Latency

We also study the implications on latency for the different implementations. Figure 2 shows the latency for 5 different scenarios using a request size of 16 B, with 15 synchronous clients and batching enabled

We would like to note that in Fig. 2 a latency-percentile diagram might not have been the best representation to get our point across. Had the experiment ran for a longer time, the differences would have not been so clear. A throughput-time diagram would have more clearly shown the effect of the crash scenarios.

The crash scenarios consists of permanently crashing the LON1 follower after 15 seconds. This should cause the leader to turn to the NYC3 server for replicating new log entries.

We would expect to see considerably higher latency for the *q=3 n=3* scenarios. However an interesting observation is that the use of batching completely masks the difference in latency. We speculate that the difference

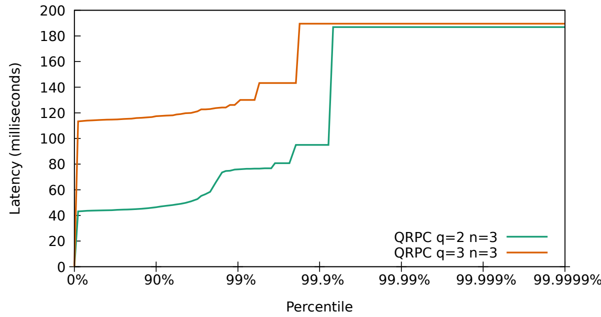

Figure 3: Latency without batching on a cluster of 3 servers. 15 clients trying to commit a request of 16 B. Clients are synchronous. Average over 5 runs.

would become more apparent under higher load. Another observation is that the latency is roughly the time between each AppendEntries RPC with some additional processing overhead, which conforms with our expectations. There are some requests that do not make it with the first AppendEntries RPC, causing higher percentiles to have a latency of double the RPC interval.

We have one outlier *QRPC w/ crash q=2 n=3*, which reveals a problem with this quorum call. Crashing the follower in LON1 forces the leader to replicate entries to the higher latency NYC3 server. This server have not been receiving log entries, and needs to be updated. This causes a dramatic increase in latency proportional to the number of missed log entries. The reason being that the implementation only cares about successful AppendEntries RPC from the fastest majority, causing the slower servers to fall behind. We discussed this problem in Sect. 3.4.

To get a better understanding of the difference in latency between the implementations, we disable batching. Figure 3 shows the latency for *QRPC q=2 n=3* and *QRPC q=3 n=3*. We do not include *No QRPC* but its latency is similar to *QRPC q=2 n=3*. Here we can clearly see the difference in commit latency, which were masked in the previous experiment. We observe that the latency for *QRPC q=2 n=3* is equivalent to the latency between NOR and LON1 with some processing overhead. Requiring a response from every server adds the additional latency of having to visit NYC3 for every request. We believe this shows the cost keeping all servers up-to-date, which avoids stalling the progress of the algorithm in the case of a change in the faster majority of the cluster.

5 Discussion

We will now discuss the benefits that we have found in using Gorums. Further we propose a non-intrusive improvement to Gorums, solving the problem we discussed in Sect. 3.2. Finally we discuss some subjects of interest for future work.

Algorithm 3 Suggested quorum function signature

```
1: func (qs QuorumSpec) QF(request Request, replies []Response)
```

5.1 Benefits

Prior to this project we have implemented network communication code for a Paxos protocol, and have experienced that it can be a hassle. When we first started the development of Raft, we were surprised to see how fast and easy it were to develop an initial prototype. This mainly comes from Gorums' *manager* abstraction handling the bookkeeping of servers and connections. The manager handles all communication within the cluster. Gorums' quorum call functionality was a big help when first implementing the RequestVote RPC, we believe it greatly reduced development time, we were also able to create tests for our quorum function easily. Quorum functions forces us to think about separation of concerns, which we believe leads to better code in the long run. It was not until we tried implementing the AppendEntries RPC that we noticed some problems with the use of quorum calls. Though these problems were mainly caused by the RPC in question, and are not inherit to Gorums.

5.2 Proposed Improvements

In response to our problem with not having access to the context in which a quorum function was initiated, we propose exposing the request that spawned the quorum function as an argument. Algorithm 3 shows how we imagine the quorum function signature would look after the change. We believe this is both easier and cleaner than having to augment RPCs arguments with additional fields, or accessing server state directly. In our case, this would allow us to inspect the server term from the request, and save us from having to add the servers term to the RequestVote and AppendEntries RPCs. As we are getting the server term from the request, when handling responses, the actual server term might have changed. This is not problematic, as the server has started a new election or discovered a higher term, in either case the work done by the quorum function will be discarded and we should return early. Though this is a proposal specific to the implementation of Raft, we believe that access to the request can be beneficial under other circumstances as well.

In the case of Raft we believe that there may be room for an additional RPC or two, if it would allow each RPC

to serve one function. We think splitting the AppendEntries RPC might not only help with the implementation of Raft in Gorums, but with the understanding of the algorithm in general. The Raft paper itself does explain the importance of decomposing problems into separate pieces that can solved, explained and understood separately. However this is not the case with Raft's AppendEntries RPC.

5.3 Future Work

In Sect. 3.7 we describe two quorum calls for handling the AppendEntries RPC. These both leave something to be desired. One is fast but suffers when the servers making up the fastest quorum change. The other quorum call's performance is dictated by the slowest server. For a future work it would be interesting to see if the use of snapshotting could offset the occasional delays in the faster quorum call.

Recently Gorums have implemented *Correctables* [4]. We believe there might be some potential to improve the Raft RPC implementations here. The reason being that Correctables allows returning intermediate values. We believe it is worth exploring.

As ViewStamped Replication and ZooKeeper Atomic Broadcast both implement a lot more messages than Raft, we are curious to find out if either of these algorithms would lend themselves better to the Gorums way of implementation. The reason being that as Raft's messages are more dense from doing more than one thing, these other algorithms might have greater separation of concerns as the mechanisms of the protocol are spread over several smaller messages.

Another experiment would be to decompose Raft's AppendEntries RPC into several isolated RPCs with separate concerns. We note that it might be hard to do this while maintaining correctness. We could then see if Raft would easier implement in Gorums. We would also be able observe if this has any effect on the understandability of the algorithm.

One thing we have not attempted is implementing reconfiguration in Raft. It would be interesting to see how much Gorums could help with reconfiguration.

6 Conclusion

In this project, we have implemented the Raft Consensus Algorithm using the Gorums framework. We wanted to figure out how useful Gorums was in this implementation, and how good a fit it were for an algorithm such as Raft. To answer this question we now draw on the experience gained through this project, and our evaluation of our Raft implementation. Overall our experience with Gorums have been good, and we believe it to be a great tool in building distributed systems. We are specially fond of how easily it handles the initial configuration of our cluster, and further communication through its *manager* abstraction. While we had some challenges with the use of Gorums' quorum calls, we believe Raft deserves some of the blame here. In this case, the AppendEntries RPC doing more than one thing. We note that while we think there is still room for innovation in the development of understandable consensus algorithms, Raft has succeeded in being easier to understand than Paxos. At least that is our opinion now, after having implemented both protocols ourselves. We conclude with Raft and Gorums not being the best match. Raft was designed to work closely with the individual responses from each server. As Gorums combines all replies into a single response, these are not easily available.

References

[1] DIGITAL OCEAN, INC. Cloud infrastructure provider. `https://www.digitalocean.com/`.

[2] GOOGLE INC. gRPC. `http://www.grpc.io/`.

[3] GOOGLE INC. Protocol Buffers. `https://developers.google.com/protocol-buffers/`.

[4] GUERRAOUI, R., PAVLOVIC, M., AND SEREDINSCHI, D.-A. Incremental Consistency Guarantees for Replicated Objects. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association.

[5] HOWARD, H., SCHWARZKOPF, M., MADHAVAPEDDY, A., AND CROWCROFT, J. Raft Refloated: Do We Have Consensus? *SIGOPS Oper. Syst. Rev. 49*, 1 (2015).

[6] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)* (2011).

[7] LAMPORT, L. The Part-time Parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998).

[8] LAMPORT, L. Paxos made simple. *ACM SIGACT News 32*, 4 (2001).

[9] LISKOV, B., AND COWLING, J. Viewstamped Replication Revisited. Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, 2012.

[10] OKI, B., AND LISKOV, B. 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)* (1988).

[11] ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm.

[12] PEDERSEN, S. M. Raft with Gorums source code. `https://github.com/relab/raft/`.

[13] Raft consensus algorithm implementations. `https://raft.github.io/#implementations`.

[14] THE GO AUTHORS. The Go Programming Language. `https://golang.org/`.

[15] TORMOD EREVIK LEA, L. J., AND MELING, H. Gorums: New Abstractions for Implementing Quorum-based Systems. In *submission* (2016).