2017 IEEE 37th International Conference on Distributed Computing Systems

Towards New Abstractions for Implementing Quorum-based Systems

Tormod Erevik Lea, Leander Jehl and Hein Meling
University of Stavanger

***Abstract*—This paper introduces Gorums, a novel RPC framework for building fault tolerant distributed systems. Gorums offers a flexible and simple *quorum call* abstraction, used to communicate with a set of processes, and to collect and process their responses. Gorums provides separate abstractions for (a) selecting processes for a quorum call and (b) processing replies. These abstractions simplify the main control flow of protocol implementations, especially for quorum-based systems, where only a subset of the replies to a quorum call need to be processed. To show that Gorums can be used in practical systems, we implemented EPaxos' latency-efficient quorum system, and ran experiments using a key-value storage. Our results show that Gorums' abstractions can provide additional performance benefits to EPaxos.**

1. Introduction

Cloud computing infrastructures offer an abundance of compute resources with different capabilities from data centers across the globe. These data centers deploy a variety of services replicated for fault-tolerance.

Replicating services requires complex protocols, and over the years, two core abstractions have been used to address this challenge, namely: the replicated state machine (RSM) abstraction [1] and the atomic storage abstraction [2]. While these abstractions have been developed from rigorous theory [3], [1], [4], [5] and implemented in a variety of prototypes [6], [7], [8], [9], and also in deployed systems [10], [11], it remains challenging to implement these abstractions correctly.

All of the above systems rely on a quorum system to achieve fault tolerance. This means that to update or access the replicated state, a process only needs to contact a quorum, e.g. a majority of the replicas. Thus, if a process contacts all replicas, but only waits for replies from a quorum/majority, the system can provide continuous service despite failures and transient outages of individual replicas. On the other hand, if processes only contact a quorum to access the replicated state, the load of these accesses can be distributed among different quorums.

The use of quorum systems presents several challenges. Firstly, a service must keep track of messages to determine whether a quorum of the replicas have replied to a request. Additional replies that arrive after a quorum of replies have been processed, must be identified and discarded.

Further, since updates can only be applied to a quorum of replicas, the replies from different replicas can diverge. It is often only possible to correctly process the content of these replies after receiving replies from a quorum. These problems become especially challenging if a process wants to dynamically change which replicas are contacted, either because the set of replicas are changing, or because the process wants to switch between contacting only a quorum and all replicas.

This paper presents Gorums, a framework for simplifying the design and implementation of fault-tolerant quorum-based systems. Gorums allows to group replicas into one or more *configurations*. Gorums enables programmers to invoke remote procedure calls (RPCs) [12] on the replicas in a configuration and wait for responses from a quorum. We call this a *quorum call*.

A quorum call not only collects replies from a quorum, it also processes these replies, combining them into a single reply. Thus, this single reply contains information about the system as a whole. For example, while a reply from any single replica may contain outdated data, replies from a quorum usually contain an up-to-date version of the data. A quorum call will only return this newest version. We use the term *quorum logic* to refer to the protocol rules for both determining whether a set of replies have been sent by a quorum, as well as how to combine these replies.

A key idea in Gorums is to clearly separate quorum logic from the main control flow of a protocol's operations. To implement a replication protocol using Gorums, the protocol's quorum logic must be implemented by a *quorum function*. This function is called by Gorums' quorum calls. If a protocol needs to use different types of quorum calls, a separate quorum function is needed for each of these. The operations of the protocol can then directly process the system state returned by the quorum calls. In our experience, the use of quorum functions helps to separate different phases of these algorithms, which usually use different quorum calls. In particular, we found that using the system state returned by quorum calls, instead of individual replies from single replicas, significantly simplified the implementation of a protocol's operations.

We have implemented Gorums as a library that exposes the quorum call and configuration abstractions to protocol clients. This library is generated from a service definition that specifies the RPCs through which replicas expose their state. Hence, a client can invoke quorum calls on a configuration to retrieve a system state. The library is generated by a tool similar to a traditional RPC stub compiler.

As a case study, we adjusted the EPaxos algorithm [7] to use Gorums, and show in §5 how the complex quorum logic

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.166

of EPaxos can be expressed using quorum functions. Finally, we also evaluated the overhead of Gorums by presenting a comparison of EPaxos, both with and without Gorums, thus measuring the performance of Gorums when used in a complete system. Our results indicate that the quorum call abstraction can improve the performance of such an implementation of a quorum-based system.

2. Quorum Systems

This section gives a short primer on quorum systems [13].

Let Π be the set of processes running a system and let $|\Pi| = n$. A *quorum* is a non-empty subset $Q \subseteq \Pi$. A *quorum system* $\mathcal{Q}$ is a collection of quorums. Every quorum system fulfills some *intersection property*, but this property may vary for different types of quorum systems.

The most common quorum system uses *majority quorums*. These quorums each contain a majority of the processes in the system ($\mathcal{Q} = \{Q \subset \Pi : |Q| > \frac{n}{2}\}$). The intersection property for this quorum system is that any two quorums intersect in at least one process.

Quorum systems are a fundamental element for building fault tolerant systems. Usually such systems are able to make progress as long as at least the processes in one quorum are available. Using majority quorums, the system can make progress if at most a minority of the processes fails. The intersection property of majority quorums ensures that no two partitions of the system can contain a quorum and make progress. This ensures that the system remains consistent. Several types of quorum systems exists. *Read-write quorum systems* and *grid quorums* are frequently used in distributed storage. Byzantine quorum systems can be used for systems that tolerate arbitrary failures.

A *latency-efficient* quorum system is composed of two quorum systems $\mathcal{Q}_f$ and $\mathcal{Q}_s$. The elements of $\mathcal{Q}_f$ and $\mathcal{Q}_s$ are called *fast* and *slow quorums*, respectively. Latency-efficient quorum systems require that all fast quorums are also slow quorums ($\mathcal{Q}_f \subset \mathcal{Q}_s$). Thus, fast quorums are usually larger than slow quorums.

EPaxos [7] uses latency-efficient quorums. In an EPaxos system with $n = 2f + 1$ processes, slow quorums are majority quorums containing at least $f + 1$ processes, while a fast quorum must contain $f + \lfloor \frac{f+1}{2} \rfloor$ processes. Both in EPaxos and other protocols, the larger fast quorums allow to commit requests in a single round-trip in the absence of contention with conflicting requests. With slow quorums, EPaxos requires two round-trips to commit such requests.

If the processes in a quorum send replies to the same request, we refer to these as a *quorum of replies*.

3. Model and Abstractions

Our environment consists of a dynamic collection of processes, where every process acts as either a protocol client, protocol server, or both. Servers expose their state via an RPC interface. Protocol clients use this interface to execute some quorum-based distributed algorithm.

A client can group several servers that provide the same RPC interface into a *configuration*. The client can then use this configuration object as a proxy for communicating with the servers by invoking the same RPC on every replica in the configuration. We call such an invocation a *quorum call*.

A quorum call does not return the replies of each individual server that sent a response to the RPC. Instead, the Gorums runtime collects these replies and passes them to a *quorum function*. Given a set of replies from some, but not necessarily all RPCs invoked as part of a quorum call, the quorum function determines: (a) whether the quorum call should return or wait for additional replies, and (b) what should be returned. When creating a configuration, a user must provide a quorum function for each RPC method provided by the servers. In the following, we explain the details of the core abstractions provided by Gorums.

3.1. Configurations and Quorum Calls

A client can combine the connections to a group of servers into a configuration, if these servers implement a common RPC interface. Quorum calls are exposed as methods on this configuration object. For every method of the RPC interface provided by the servers, a configuration provides a quorum call with the same signature.

A configuration is local to the client and thus different clients may have different configurations. Since configurations are local, their creation does not require any network operations, and a server may not be aware that it is included in a configuration. To communicate with different, but not necessarily disjoint groups of servers, a client may create as many configurations as needed. Configurations are immutable, and new configurations are created by clients on demand, rather than changing the state of the configurations themselves. We have used this to implement several reconfiguration algorithms for atomic storage [14].

Many systems [15] implement complex protocols to establish and maintain a synchronized global configuration, or membership view, among the servers. Gorums does not synchronize its configurations. This design enables significant flexibility in how clients interact with the servers. For instance, a client can communicate with all processes in some approximation of the global configuration, or it can use another configuration to communicate with a preferred subset of the processes in the global configuration. Moreover, a client can also use two different configurations concurrently, e.g. to represent an old and new global configuration during a transition period.

3.2. Quorum Specification Objects

When creating a new configuration, the user must provide a quorum function for each RPC method on the configuration. This is accomplished by implementing the quorum functions as methods on a *quorum specification object*, or QuorumSpec.

The QuorumSpec specifies the static properties of a quorum system. In simple examples it is enough if a QuorumSpec, *qs*, contains a single parameter *qs*.QSize, specifying how

many of the processes in a configuration are necessary to form a quorum. However, in many cases the QuorumSpec may also store more complex information, e.g. for Byzantine quorum systems the QuorumSpec may store cryptographic keys needed to verify message content.

3.3. Quorum Functions

A quorum call returns a single reply, masking the fact that the RPC call is actually invoked on several servers, which may return multiple replies.

To determine when and what to return, Gorums uses a user-defined quorum function. The quorum function is invoked whenever a new reply is received, and it has two responsibilities. First, to determine if a quorum of replies have been collected, and second, to combine individual replies into one. If this combination results in a valid return value for the quorum call, Gorums will terminate outstanding individual RPCs and return this value.

While quorum functions can usually be implemented with only a few lines of code, they hold the core logic of fault tolerant distributed algorithms. The decision whether a given set of replies is sufficient for a quorum call to return, actually defines the fault tolerance of a system. Similarly, by processing and combining individual replies, a quorum function extracts information about the system state from the states of individual servers.

A key advantage of user defined quorum functions is that a quorum function can access the full content of replies. This allows more complex logic than simply waiting for replies from a majority of the servers. For example, a quorum function can detect conflicting or inconsistent replies and either abort the call early, or wait for additional replies to try and resolve the conflict.

3.4. Quorum Call Semantics

As explained in §3.1, for every RPC Method(*args*) provided by the servers, Gorums defines a corresponding method on the configuration object c:

$$\text{MethodReply, Error} := c.\text{Method}(args)$$

This quorum call will invoke s.Method(*args*) on all servers in c, and Gorums will process the replies using the quorum function MethodQF():

$$\text{MethodReply, Bool} := qs.\text{MethodQF}(replies\ [\,]\text{MethodReply})$$

This quorum function is invoked every time an individual RPC s.Method() returns. The *replies* argument is a collection of all individual replies received so far.

The quorum function returns a single MethodReply and a boolean. The boolean signals to Gorums whether it can return from the quorum call, or should wait for more replies. When the c.Method() returns, it returns the MethodReply returned by the last call to MethodQF().

A quorum call will also return if all individual RPCs have either completed without satisfying the requirement expressed in the quorum function, or returned an error. In these cases, a quorum call will return an *IncompleteError*, and include the reply returned by the last invocation of its quorum function. Table 1 summarizes the relation between the return values of the quorum function and the behavior of a quorum call.

TABLE 1. Quorum call behavior

#	**QFunc return** Reply, Bool	**Quorum call action** return Reply, Error
1	*retval*, *true*	return *retval*, *nil* and terminate call
2	*retval*, *false*	if possible: wait for further replies else: return *retval*, *IncompleteError*

4. Using Quorum Functions

In this section we first give a template for implementing quorum functions, followed by two examples that illustrate the different aspects of the template.

4.1. Quorum Function Template

Below we show a template for quorum functions, with all the functionality that we have found necessary for different quorum systems. The template has six parts, indicated with line numbers. Each part implements different functionality and many quorum functions do not require all six.

Template for quorum functions

```
0: func (qs QuorumSpec) MethodQF(replies []MethodReply)
1:     if qs.Abort(replies[len(replies) − 1]) then          ▷ check last reply
           return replies[len(replies) − 1], true           ▷ abort call
2:     if len(replies) < qs.QSize then                      ▷ check quorum size
           return nil, false          ▷ no quorum yet, await more replies
3:     if ¬qs.IsQuorum(replies) then                        ▷ check content for quorum
           return nil, false          ▷ no quorum yet, await more replies
4:     reply := qs.Combine(replies)                         ▷ combine replies into single reply
5:     if qs.WaitForMore(replies) then
           return reply, false        ▷ return possible, but prefer waiting
6:     return reply, true                                   ▷ terminate call and return
```

Part 1 calls an Abort() function that only inspects the last reply and determines whether the quorum call should be aborted. The last reply in the array passed to a quorum function is always the last reply that was received. If the argument to the quorum function includes several replies, all but the last have been submitted to Abort() during previous invocations of the quorum function. If Abort() returns *true*, the quorum function also returns *true*, causing the quorum call to terminate. The quorum call will return the reply that caused the abort, and the caller can identify the abort by inspecting this reply. Part 1 can be omitted for quorum calls that do not abort.

The second part compares the number of replies to a parameter QSize that is stored in the QuorumSpec. This parameter specifies the minimum number of replies necessary to form a quorum. If fewer than QSize replies have been

received, the quorum function returns *false*, causing the quorum call to wait for additional replies.

For many quorum systems, e.g. majority quorums, comparing the number of replies in Part 2 is enough to determine whether a quorum was found. However, for more complex quorum systems, e.g. grid quorums or Byzantine quorum systems, more complex checks that accesses the actual content of replies are necessary. These are performed in Part 3.

In Part 4, the content of the individual replies is combined into a single reply that can be returned by the quorum call. This part is present in most quorum functions, except simple examples, where replies are only acknowledgments and carry no other content.

In Part 5, the quorum function can wait for additional replies, even though a quorum was already received. Most examples omit this part. However, in latency-efficient quorum systems, Part 5 can be used to wait for a fast quorum, when a slow quorum was already received. We do this in our EPaxos implementation (see §5). Note that Part 5 is the only case in our template where the quorum function returns a non-nil reply together with a boolean *false*. The reason for this is that a quorum of replies is already present, which allows to determine a correct return value.

Finally, in Part 6 the combined reply is returned together with a boolean, indicating to Gorums that the quorum call can be terminated.

We note that the template only serves as a guideline. Advanced quorum functions typically deviate from this template. In particular, to avoid repeated iterations over the *replies* array, it is common to combine Parts 3, 4, and 5.

4.2. Example Quorum Functions

In the following we present two simple quorum functions derived from well-known distributed algorithms, namely distributed storage and distributed consensus. This aims to show that Gorums is applicable to a wide range of protocols that depend on multiple replies.

Choosing a reply: The quorum function in Algorithm 2 implements an essential part of the read operation of distributed storage. In algorithms for distributed storage, values are usually stored together with a timestamp to distinguish old values from newer values [4], [8], [5]. A read operation has to collect timestamped values from a quorum of servers and return the value with the highest timestamp. This selection is performed by ReadQF(). Algorithm 2 only implements Parts 2, 4 and 6.

In this quorum function, it is worth noting that we need not inspect the replies until we have collected ReadQSize replies, which is necessary to form a read quorum. Further, when inspecting replies in Part 4, we need not combine them, only choose the one with the highest timestamp.

Combining replies: We next illustrate Part 4 with a specific example in Algorithm 3. The PaxosPrepareQF quorum function implements the quorum logic of the first phase of the Paxos consensus algorithm [3]. This algorithm is a key component in many fault tolerant systems. Our objective here is not to explain the intricacies of the algorithm, but to highlight that this is an instance of combining replies. Specifically, we first create an empty reply on Line 4, and then populate it with information from the received replies according to phase 1 of Paxos. This is necessary because each individual reply may not have enough information.

Algorithm 2 Atomic storage read quorum function

```
1: func (qs QuorumSpec) ReadQF(replies []ReadReply)
2:     if len(replies) < qs.ReadQSize then          ▷ read quorum size
           return nil, false                        ▷ no quorum yet, await more replies
3:     reply := withHighestTimestamp(replies)
4:     return reply, true
```

Algorithm 3 Paxos first phase quorum function

```
1: func (qs QuorumSpec) PaxosPrepareQF(replies []Promise)
2:     if len(replies) < qs.majQSize then           ▷ majority quorum size
3:         return nil, false                        ▷ no quorum yet, await more replies
4:     reply := new(Promise)                        ▷ initialize reply with nil/0 fields
5:     for r := range replies do
6:         if r.ballot > reply.ballot then
7:             reply.ballot := r.ballot
8:         if r.vballot ≥ reply.vballot then
9:             reply.vballot := r.vballot
10:            reply.value := r.value
11:    return reply, true                           ▷ quorum found
```

5. Case Study: Latency-efficient Quorums

As a case study, we implemented EPaxos [7], which uses latency-efficient quorums; see §2. In the following, we focus on a quorum function that uses both fast and slow quorums.

EPaxos is a fairly new protocol for state machine replication. Instead of using a single leader, EPaxos allows all replicas to propose commands. That is, each replica becomes the leader for its own commands. If concurrently proposed commands are not conflicting, all commands can be committed after a single (pre-accept) round. If different replicas propose conflicting commands, these have to be committed in an additional (accept) round. With this approach, EPaxos achieves low latency in wide area networks, and avoids the single leader bottleneck.

In EPaxos every replica is both a leader and maintains state. Thus, every replica needs to implement both a protocol client used by the leader, and a protocol server that allows other leaders to access the replicated state via quorum calls. This is different from implementations of reconfigurable atomic storage, where clients (that perform operations) and servers (that replicate state) are decoupled and only communicate via quorum calls.

In our implementation of EPaxos, the leader invokes quorum calls on a configuration that only includes the other replicas, and accesses its local state directly. Thus, a leader receives an implicit reply from itself, and we had to adjust the QSize parameters in our quorum function accordingly.

In the following we focus on the pre-accept phase of EPaxos, which uses both fast and slow quorums. This pre-accept phase is similar to the first phase of two-phase com-

Algorithm 4 EPaxos PreAccept quorum function

```
1: func (qs QuorumSpec) PreAcceptQF(replies []PreAccReply)
2:     if replies[len(replies) − 1].Type = Abort then
3:         return replies[len(replies) − 1], true                ▷ single Abort
4:     if len(replies) < qs.SlowQSize then
5:         return nil, false                                     ▷ no quorum yet
6:     reply := new(PreAccReply)                                 ▷ initialize reply with nil/0 fields
7:     for r := range replies do
8:         if r.Type = Conflict then
9:             reply.Type := Conflict
10:            reply.Conflicts := reply.Conflicts ∪ r.Conflicts
11:    if (reply.Type = Ok) ∧ (len(replies) < qs.FastQSize) then
12:        reply.Type := Conflict
13:        return reply, false
14:    return reply, true
```

mit. A leader forwards a new state update to all servers in a PreAccept message. These servers reply with a PreAccReply.

Our PreAccept quorum function in Algorithm 4 processes PreAccReply messages. These messages can be one of three types, Abort, Conflict, and Ok.

If a leader receives a PreAccReply of type Abort, it is no longer a leader, and can therefore abort outstanding quorum calls. This correspond to Part 1 of our Template.

A PreAccReply of type Conflict contains information about a conflicting request that was proposed or committed by a different leader. When combining replies (Part 4 of our Template) our quorum function adds all reported conflicts to the return value.

Lines 11-13 implement Part 5 of the Template. Here, the quorum function decides whether it should wait for additional replies. In EPaxos, a leader can commit a request after the pre-accept phase only if it received replies of type Ok from a fast quorum. We therefore only wait for a fast quorum if none of the replies are of type Conflict.

Note that, if any replies of type Conflict have been received, or if the existing replies are not enough to form a fast quorum, the return value will also be of type Conflict. Thus, our quorum function only returns a reply of type Ok if it received replies of type Ok from a fast quorum.

6. Implementation

We have implemented Gorums as a library in Go [16]. The source code will be freely available [17]. Gorums uses code generation to produce an RPC library that clients can use to invoke quorum calls. Our code generation tool builds on an existing toolchain consisting of Protocol Buffers (protobuf) [18] and gRPC [19].

Protocol messages and service APIs for Gorums are specified using the standard interface description language supported by protobuf. Our code generator is implemented as a second order plugin to the protobuf compiler toolchain. The generator adds a layer on top of the regular gRPC API, exposing Gorums's API necessary for clients to perform quorum calls and administer configurations.

As for a regular gRPC implementation, developers must implement the server-side RPC methods specified in the gRPC interfaces. Additionally for Gorums, developers must supply a client-side quorum function for each of the RPC methods, implementing the desired quorum logic for the protocol being implemented.

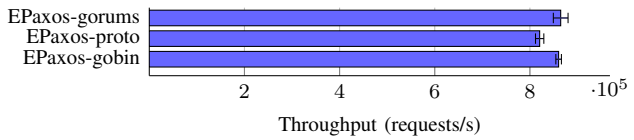

Figure 1. Throughput for 16 B commands and 5 replicas. Average over 20 runs with 95 % confidence interval.

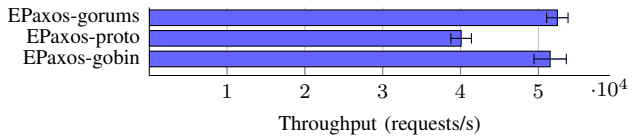

Figure 2. Throughput for 1 kB commands and 5 replicas. Average over 20 runs with 95 % confidence interval.

7. Evaluation

We have modified the original EPaxos implementation [20] to evaluate the performance impact of using our Gorums framework in a sophisticated RSM implementation.

The original implementation of EPaxos uses TCP message passing and a specialized serialization library, gobin [21]. However, Gorums is based on gRPC and uses protobuf for serialization. For this reason, we have also created a modified version of EPaxos [22] that instead uses protobuf for message serialization. We have done this to illustrate the impact of encoding choice, in addition to be able to more fairly compare our results. This version therefore serves as our main baseline when evaluating Gorums. Summarized, our benchmarks uses the following three versions of the original EPaxos implementation:

EPaxos-gobin	TCP with gobin serialization (original)
EPaxos-proto	TCP with protobuf serialization
EPaxos-gorums	Gorums with protobuf serialization

We repeat one benchmark from the original EPaxos description [7] for a key-value store in a LAN setting to evaluate system throughput for the three different versions listed above. A single client on a separate machine sends write requests in an open loop to mimic an unbounded number of clients and to saturate the system. Each command leader batches all requests in its queue, up to a maximum batch size of 1000 requests. All benchmarks uses 5 replicas and a 2 % command interference rate. This means that 2 % of client requests target the same key. EPaxos is forced to invoke its slow-path if concurrent write requests touch the same key. We measure the steady-state throughput for the commit protocol with no replica failures.

Figure 1 shows the throughput for client requests of size 16 B. EPaxos-proto has a 5 % overhead compared to EPaxos-gorums. The main reason that EPaxos-gorums performs better, is that EPaxos-proto (and EPaxos-gobin)

perform more work in their dispatch loop. That is, they decode every message received, even if the message is a response to a call for which the server has already received replies from a quorum. EPaxos-gorums discards these redundant messages, relieving the EPaxos dispatch loop from useless work. For this benchmark however, EPaxos-gobin performs on a par with EPaxos-gorums due to its specialized serialization library.

Figure 2 shows the throughput for a client request size of 1 kB. For this request size EPaxos-gorums achieves around 30 % higher throughput than EPaxos-proto, and now also outperforms EPaxos-gobin by approximately 2 %. In this benchmark, significantly more time is spent encoding and decoding protocol messages compared to the benchmark with a 16 byte request size. This further increases EPaxos-gorums' advantage of being able to discard messages before decoding and to avoid processing them.

8. Related Work

Similar to our quorum call abstraction, several works [23], [24] uses the term Q-RPC to express the invocation of RPCs on several servers and collecting replies from a quorum. The term is used for describing protocols, and it is not reported if it is used as an abstraction in any implementation. To our knowledge, no experimental evaluation of the Q-RPC approach exists. The Q-RPC approach only collects responses from a quorum, while our quorum calls also combine replies.

Today there exists a wide variety of RPC frameworks [19], [25], but most of these do not offer any support for transparently invoking a group of servers and collecting a quorum of responses. Remote Method Invocation (RMI) is the object-oriented equivalent to RPC, and is commonly used in CORBA- and Java-based distributed applications. Several researchers have extended RMI with the ability to invoke a group of servers [26], [27]. These efforts have largely been focused on simple majority quorums. Recent versions of JGroups [27] have added the ability to specify and use a response filter function for RPCs, similar to our quorum function. Moreover, most of the above frameworks have been developed within the confines of an underlying group communication system and its membership service.

9. Conclusion

In this paper, we have presented Gorums, a framework meant to aid the design and implementation of quorum-based systems. Gorums provides a quorum call abstraction which reduces the complexity of building clients that communicate with quorum systems. We use quorum functions to encapsulate quorum logic. Using Gorums, the main control flow of protocol operations can be implemented based on the system state computed by our quorum functions. The examples included in this paper show that this approach is applicable to various and complex protocols. While the fundamental idea of quorum call is simple, we believe the abstraction strikes a balance between encapsulating low-level details and exposing the necessary functionality to implement a wide range of distributed protocols.

References

[1] F. B. Schneider, "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, Dec. 1990.

[2] L. Lamport, "On Interprocess Communication," 1985.

[3] ——, "The Part-time Parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, May 1998.

[4] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message-passing Systems," *J. ACM*, vol. 42, no. 1, Jan. 1995.

[5] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic Atomic Storage Without Consensus," *J. ACM*, vol. 58, no. 2, Apr. 2011.

[6] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *USENIX ATC*, 2014.

[7] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is More Consensus in Egalitarian Parliaments," in *SOSP*, 2013.

[8] S. Gilbert, N. A. Lynch, and A. A. Shvartsman, "Rambo: A Robust, Reconfigurable Atomic Memory Service for Dynamic Networks," *Distributed Computing*, vol. 23, no. 4, 2010.

[9] L. Jehl, R. Vitenberg, and H. Meling, "SmartMerge: A New Approach to Reconfiguration for Atomic Storage," in *DISC*, 2015.

[10] M. Burrows, "The Chubby Lock Service for Loosely-coupled Distributed Systems," in *OSDI*, 2006.

[11] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX ATC*, 2010.

[12] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, Feb. 1984.

[13] M. Vukolic, *Quorum Systems: With Applications to Storage and Consensus*. Morgan and Claypool, 2012.

[14] L. Jehl and H. Meling, "The Case for Reconfiguration without Consensus," in *OPODIS*, 2016.

[15] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group Communication Specifications: A Comprehensive Study," *ACM Comput. Surv.*, vol. 33, no. 4, Dec. 2001.

[16] The Go Authors. The Go Progarmming Language. [Online]. Available: http://www.golang.org

[17] T. E. Lea, L. Jehl, and H. Meling. Gorums code base. [Online]. Available: http://www.github.com/relab/gorums

[18] Google Inc. Protocol Buffers. [Online]. Available: http://developers.google.com/protocol-buffers

[19] ——. gRPC. [Online]. Available: http://www.grpc.io

[20] I. Moraru. EPaxos code base. [Online]. Available: http://www.github.com/efficient/epaxos

[21] Efficient Computing at Carnegie Mellon. gobin-codegen. [Online]. Available: https://github.com/efficient/gobin-codegen

[22] T. E. Lea. EPaxos-Gorums code base. [Online]. Available: http://www.github.com/relab/epaxos

[23] D. Malkhi and M. K. Reiter, "An Architecture for Survivable Coordination in Large Distributed Systems," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 2, 2000.

[24] L. Gao, M. Dahlin, J. Zheng, L. Alvisi, and A. Iyengar, "Dual-Quorum Replication for Edge Services," in *Middleware*, 2005.

[25] Apache Software Foundation. Apache Thrift. [Online]. Available: https://thrift.apache.org

[26] H. Meling, A. Montresor, B. E. Helvik, and O. Babaoglu, "Jgroup-ARM: A Distributed Object Group Platform with Autonomous Replication Management," *Softw. Pract. Exper.*, vol. 38, no. 9, Jul. 2008.

[27] RedHat Inc. JGroups – A Toolkit for Reliable Messaging. [Online]. Available: http://jgroups.org/