FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme / specialisation: Computer Science	Spring semester, 2024 Open / ~~Confidential~~
Author: Aleksander Vedvik	
Supervisor: Hein Meling	
Thesis title: All-to-all Communication for Multiparty RPC	
Credits (ECTS): 30	
Keywords: Distributed Systems, Quorum Systems, Gorums, RPC, Consensus, All-to-all communication, Broadcast, State Machine Replication, Paxos, PBFT	Pages: 115 + code delivered as an embedded zip file. Stavanger, June 15, 2024

ALEKSANDER VEDVIK

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

All-to-all Communication for Multiparty RPC

Master's Thesis - Computer Science - June 2024

```
func (m *Manager) NewConfiguration(opts ...gorums.ConfigOption) (c *Configuration, err error) {
    if len(opts) < 1 || len(opts) > 2 {
        return nil, fmt.Errorf("wrong number of options: %d", len(opts))
    }
    c = &Configuration{}
    for _, opt := range opts {
        switch v := opt.(type) {
        case gorums.NodeListOption:
            c.Configuration, err = gorums.NewConfiguration(m.Manager, v)
            if err ≠ nil {
                return nil, err
            }
        case QuorumSpec:
            // Must be last since v may match QuorumSpec if it is interface{}
            c.qspec = v
        default:
            return nil, fmt.Errorf("unknown option type: %v", v)
        }
    }
    // return an error if the QuorumSpec interface is not empty and no implementati
    var test interface{} = struct{}{}
    if _, empty := test.(QuorumSpec); !empty && c.qspec == nil {
        return nil, fmt.Errorf("missing required QuorumSpec")
    }
    return c, nil
}
```

I, **Aleksander Vedvik**, declare that this thesis titled, "All-to-all Communication for Multiparty RPC" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master's degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- I have used ChatGPT and Grammarly solely for assistance with sentence structuring and wording purposes in this thesis. All ideas, research, and content are my own or clearly attributed as stated above, and the use of these tools were limited to refining the clarity and coherence of the text. The final content has been reviewed and approved by me to ensure accuracy and integrity.

“Programming is a nice break from thinking.”

– Leslie Lamport

Abstract

Quorum systems, like replicated state machines, are essential in distributed systems, but building correct and high-performance systems is challenging due to the asynchronous environments they operate in. Gorums is a novel framework that simplifies the creation of quorum-based systems, alleviating developers of the complexities associated with building fault-tolerant services. However, it currently lacks support for all-to-all communication, which is crucial for robust inter-node communication and is often used in consensus algorithms like Practical Byzantine Fault Tolerance (PBFT) and Paxos.

This thesis analyzes the design requirements needed to facilitate all-to-all communication in quorum-based systems. We extend Gorums with a new broadcasting framework, introducing an easy-to-use abstraction named BroadcastCall. BroadcastCall is an RPC invocation to a group of servers communicating in an all-to-all pattern. Furthermore, it relates individual messages sent by the servers to a single abstraction called broadcast request. To evaluate the broadcasting framework, we implement both PBFT and Paxos using the framework and compare them to baseline implementations. The results demonstrate that the framework delivers performance comparable to the baseline implementations without incurring significant overhead.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Hein Meling, for his enthusiasm, invaluable feedback, and unwavering support throughout my work on this thesis.

I am also grateful to the Relab team—Hein Meling, Leander Jehl, Hanish Gogada, Arian Baloochestani Asl, Jostein Hagen Lindhom, Jayachander Surbiryala, and Nejm Saadallah—for the weekly meetings, discussions, and insights. Your collective support has been instrumental in steering my research in the right direction.

Contents

Nomenclature

broadcast framework The new functionality extended from the current version of Gorums.

user The user implementing a protocol or algorithm using Gorums.

client A process initiating and sending requests to a set of servers.

server A process responsible for handling and replying to requests.

node Either a server or a client.

replica Mainly a server but can also act as a client.

Gorums configuration An object representing a client.

Gorums client Synonym for Gorums configuration.

Gorums server The server generated by Gorums.

view A Gorums configuration that is used by a server.

o Origin: A node issuing broadcast requests.

B Broadcast request: A request sent from an origin that spans several servers and handlers.

m Message: a single request sent between a client-server or server-server pair. Messages are always related to a unique broadcast request.

P Broadcast processor.

θ The state corresponding to a broadcast processor.

$\mathcal{C}$ The set of clients in a system.

$\mathcal{S}$ The set of servers in a system.

π A subset of the servers in a system.

$\mathcal{A}$ The set of actions that can be performed on the broadcast processor.

ν A specific action, e.g. *Broadcast*.

$\mathcal{H}$ The set of all RPC methods defined in the proto specification.

$\mathcal{H}_B$ The set of all RPC methods defined in the proto specification that uses the broadcasting functionality.

h A single RPC method that is defined in the proto specification.

ψ The order index that is used to facilitate execution ordering.

$\mathcal{M}$ The set of all messages sent between servers and clients.

$\mathcal{M}_B$ The set of all messages related to a specific broadcast request.

Chapter 1

Introduction

1.1 Background and Motivation

Historically, distributed computing was expensive, complex, and difficult to manage. Today, distributed computing is used in diverse fields, from database management to video games, and is essential for modern technologies like cryptocurrency systems, scientific simulations, blockchain, and AI platforms. Distributed systems handle workloads too large for a single device and adapt to changing demands, such as surges in e-commerce traffic. They enable advanced features like off-site backups and power everyday tasks like sending emails and browsing the web [1]. Distributed computing is offered as high-availability services or as part of cloud infrastructure solutions like Kubernetes [2], Docker Swarm [3], Apache Kafka [4], RabbitMQ [5], Azure Service Bus [6], and AWS Simple Queue Service [7]. These services typically include a control plane that uses consensus mechanisms to replicate nodes and create clusters, ensuring reliability and scalability [8].

Designing and building distributed systems that are fault-tolerant and strongly consistent is inherently challenging [9][10]. Achieving high availability often necessitates replacing failed servers without disruption, adding to the complexity of development and maintenance [11, p. 17][12, p. 1]. Thus, it is vital to equip developers with tools that provide higher-level abstractions to mitigate this complexity and streamline the development process. Quorum systems, like replicated state machines, are essential but challenging due to non-deterministic state changes and complex protocols [13]. Synchronizing activities amid partial failures and adversarial attacks is difficult due to asynchronous processes and possible communication failures [12, p. 1]. Replication ensures data availability despite failures, but maintaining consistency and robustness in such systems remains a significant challenge due to the asynchronous, heterogeneous, and failure-prone nature of the environment [14, p. 1][15].

Gorums is a Go-based framework designed to simplify the creation of quorum-based systems, though it currently lacks support for all-to-all communication between processes [16]. While Go [17] provides robust concurrency primitives, the combination of traditional locking and channel-based message passing can lead to specific bugs, notably communication deadlocks, which account for over 50% of deadlock bugs in Go applications [18][19]. Distributed system algorithms often rely on consensus for correct execution, built on abstractions like reliable message delivery and node connectivity [20]. However, modern systems face errors and failures, necessitating timeouts, retries, proper TCP connection handling, and cleanup of stale data to prevent memory errors [21]. Asynchronous coding, requiring advanced concurrency mechanisms to prevent race conditions, can potentially cause performance issues under load [22]. Thus, careful design is essential for scalability and performance, considering protocol abstractions such as timeouts, message ordering, and delivery guarantees. [23][24]

In parallel or distributed computing systems, processes often need to communicate with each other [25]. Many systems and algorithms, such as graph algorithms like PageRank [26] and graph partitioning [27], or parallel sorting algorithms [28], rely on the ability to send and receive messages among all processes. Open-source applications like Apache Spark [29] also utilize all-to-all communication, as seen in the shuffle phase in Fig. 1.1. All-to-all communication is crucial because it allows robust communication among distributed processes, enabling each node to communicate with every other node. This pattern is essential for consensus algorithms such as Practical Byzantine Fault Tolerance (PBFT) [20] and Paxos [30].

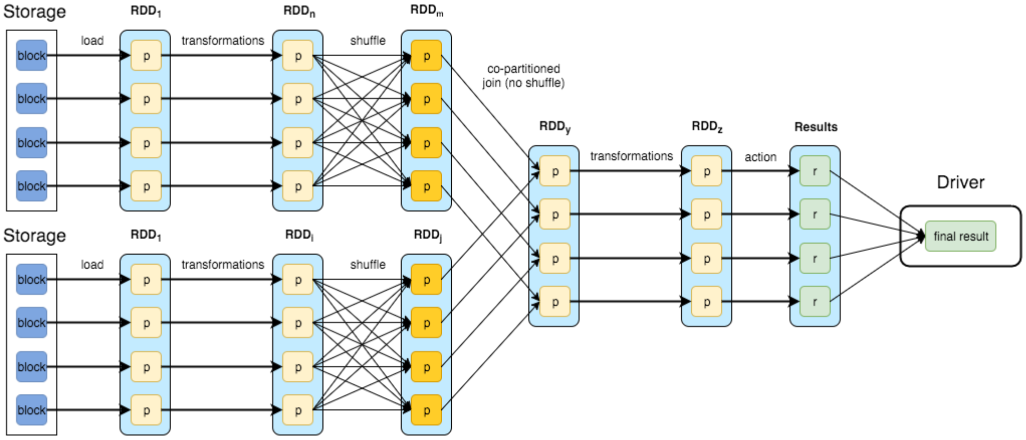


Figure 1.1: An example DAG created by Apache Spark. The data processing workflow is defined as reading the data source, applying a set of transformations, and materializing it into the final result. The DAG utilizes the all-to-all communication pattern in some of the stages. Photo retrieved from [31].

1.2 Objectives

The objectives of this thesis are encapsulated in two research questions:

- **Research question 1:** Is it possible to create an easy-to-use framework for quorum-based systems that enables the all-to-all communication pattern?

- **Research question 2:** Is it possible to make it reliable and secure without creating too much overhead?

1.3 Approach and Contributions

The contributions of this thesis are as follows:

- We conducted an analysis of various algorithms that utilize the all-to-all communication pattern and developed a module specification based on this analysis.

- We developed a framework named broadcast framework to facilitate the all-to-all communication pattern, integrating it with Gorums and extending Gorums with additional functionality.
- We introduced an abstraction called BroadcastCall, analogous to QuorumCall in Gorums, but enabling all-to-all communication.
- We conducted a security analysis of the broadcast framework, provided mitigation strategies, and highlighted its shortcomings.
- We implemented both process-centric and data-centric versions of Paxos using the broadcast framework.
- We implemented PBFT using the broadcast framework.
- We implemented Eager Reliable Broadcast using the broadcast framework.
- We evaluated the broadcast framework by using the implementations of Paxos and PBFT and comparing them to baseline implementations in terms of common case operations and throughput versus latency.

1.4 Outline

The structure of the remainder of this thesis is as follows:

Chapter 2 introduces relevant background material, including Gorums, Paxos, and PBFT.

Chapter 3 presents related work, covering systems that assist in building distributed applications, algorithms for implementing quorum-based systems, and frameworks for enabling all-to-all communication.

Chapter 4 provides an extensive analysis identifying the challenges and design requirements of implementing abstractions for all-to-all communication. It also outlines the design decisions and creates a module specification. Finally, the architecture of the new framework is presented.

Chapter 5 describes the implementation and its integration with Gorums. It offers an in-depth description of how the implementation adheres to the module specification created in Chapter 4.

Chapter 6 demonstrates a practical implementation of a broadcasting algorithm using the new framework.

Chapter 7 provides an experimental evaluation of the new framework compared to baseline implementations of Paxos and PBFT. It assesses the overhead in common case operations and compares the relative throughput and latency of each algorithm.

Chapter 8 concludes the thesis, discussing limitations and areas for further work.

Chapter 2

Background

This chapter provides the background material relevant to this thesis. We begin by introducing the core concepts in distributed systems, followed by a presentation of Gorums. The chapter concludes with descriptions of both PBFT and Paxos.

2.1 Distributed systems

Distributed systems involve algorithms designed for a set of processes working together to achieve a common goal. These processes operate concurrently and may experience partial failures, such as crashes or disconnections, while others continue to function. This very notion of partial failures distinguishes distributed systems from concurrent systems. The primary challenge is ensuring that the remaining active processes can synchronize their activities in a consistent way despite failures and potential adversarial attacks. Distributed computing encompasses both client-server and multiparty interactions, where multiple processes must coordinate to achieve robust, reliable communication and task execution. This complexity makes distributed computing a challenging yet essential aspect of modern applications like web browsing [11, p. 17].

2.1.1 System Model

We focus on distributed systems used in blockchains or for state machine replication (SMR), requiring a fault-tolerant message-passing system that executes requests. We assume a system where processes can be clients or servers, with clients and servers sending requests and servers processing them. The system must be quorum-based, with processes being correct, faulty, or Byzantine, and operating in a partially synchronous environment. Additionally, we assume the adversary is computationally bounded and unable to break cryptographic techniques such as digital signatures or transport layer security (TLS).

2.1.2 Quorum Systems

Quorum systems, such as replicated state machines, are essential abstractions in distributed fault-tolerant computing. They capture trust assumptions and form the backbone of many algorithms for reliable broadcasts, shared memory, and consensus [13][16][32]. Progress is allowed as long as at least one quorum is available. Majority quorums, where only a minority of processes can fail, ensure system consistency by requiring that no two partitions can each contain a quorum and make progress. Various types of quorums exist, including read-write quorums and grid quorums for distributed storage, and Byzantine quorum systems for tolerating arbitrary failures [16]. Byzantine quorum systems are essential for constructing resilient distributed systems from untrusted components, as they handle potentially malicious nodes and colluding groups of nodes. Traditionally, these systems assume symmetric trust, with a global assumption about which nodes may fail. However, trust is subjective and can vary, necessitating models that accommodate asymmetric trust assumptions [32].

2.1.3 Consensus

Consensus is a critical concept in distributed computing, serving as a fundamental abstraction that addresses the challenge of achieving agreement among multiple processes on a common value, despite unreliable communication and faulty processes [11, p. 274][33]. Consensus protocols typically assume a limited number of faulty processes and a shared, symmetric trust assumption among all processes. However, with the rise of blockchain systems like Ripple [34] and Stellar [35], more flexible trust models have emerged. These models allow each process to specify its own trusted processes, providing greater flexibility than traditional assumptions that only limit the number of faulty processes [33].

At its core, consensus can be described as processes agreeing on a common value out of values they initially propose [11, p. 274]. Consider a group of processes that can propose values. A consensus algorithm ensures that only one of the proposed values is chosen, and if no value is proposed, then no value should be chosen. Once a value is chosen, processes must be able to learn the chosen value. The safety requirements for consensus are as follows [11, p. 275][30]:

- **Validity**: Only a proposed value can be chosen.
- **Agreement**: No two processes should decide on different values.
- **Termination**: Every correct process eventually decides on a value.
- **Integrity**: No process decides twice.

2.1.4 Group Communication

In a parallel or distributed computing system, processes frequently need to communicate with one another [25]. A group is a collection of interconnected processes with an abstraction layer that hides the underlying message passing between them, making the communication appear as a normal procedure call to a single, non-replicated remote server object [36][23]. View-oriented group communication is a crucial and widely used component in many distributed applications. Several proposals address this by extending distributed objects into distributed object groups [23][37]. Depending on the number of processes involved, communication can take several forms: one-to-one, one-to-many, one-to-all, and all-to-all [25].

Unicast is a one-to-one type of communication between processes [25][36]. Fig. 2.1 shows an example of *unicast* communication between two processes in a system. It also illustrates an example of one-to-many communication, known as *multicast*, where a process communicates with a subset of processes [25][36].

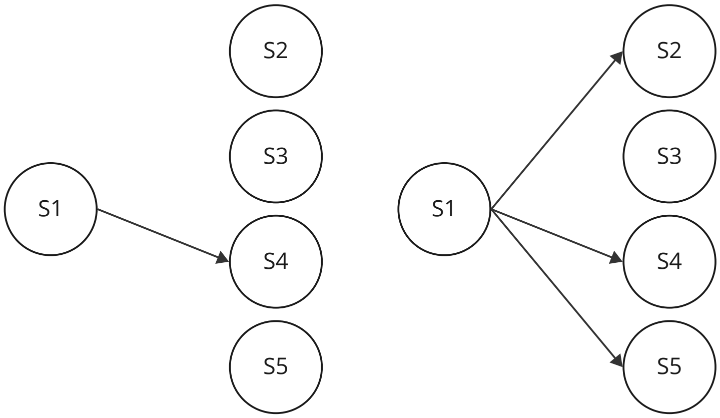


Figure 2.1: *Left*: Unicast (one-to-one) communication: S1 is communicating with a single process in the system. *Right*: Multicast (one-to-many) communication: S1 is communicating with a subset of processes in the system.

Broadcast communication is useful when a process attempts to communicate with all processes in a distributed system simultaneously [36]. Fig. 2.2 shows an example of *one-to-all broadcast* and *all-to-all broadcast*. In *all-to-all broadcast*, every process in a group sends a message to all the other processes [25].

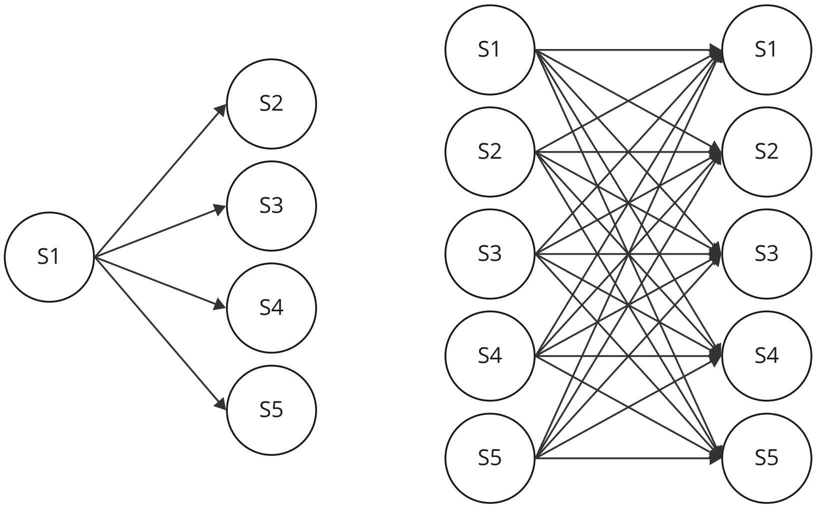


Figure 2.2: *Left*: Broadcast (one-to-all) communication: S1 is communicating with all processes in the system. *Right*: Broadcast (all-to-all) communication: all processes are communicating with all processes in the system.

2.2 Gorums

Gorums is a framework designed to simplify the design and implementation of fault-tolerant quorum-based systems and is built atop the Remote Procedure Call (RPC) framework gRPC and using the Go programming language. It allows grouping replicas into one or more configurations. This enables programmers to invoke remote procedure calls (RPCs) on the replicas within a configuration and wait for responses from a quorum [16].

2.2.1 Technology

Gorums is built using the Go programming language [17] and the Remote Procedure Call (RPC) framework gRPC [38]. Go is an open-source language designed with concurrency in mind, making it easy to write programs that maximize the use of multicore and networked machines. It is expressive, concise, clean, efficient, statically typed, and compiled. Additionally, Go features a fast garbage collector and run-time reflection [17]. Go includes two built-in concurrency constructs called goroutines and channels [39]. Goroutines are lightweight threads managed by the Go runtime, while channels provide CSP-style [40] message-passing mechanisms for communication between goroutines. Furthermore, Go offers a suite of concurrency primitives such as mutexes, condition variables, and atomic operations, facilitating both shared-memory and message-passing concurrency [18, p. 187-188].

The gRPC framework is an open-source, high-performance RPC framework designed to operate in any environment. It efficiently connects services within and across data centers, offering pluggable support for load balancing, tracing, health checking, and authentication. Additionally, gRPC is well-suited for the last mile of distributed computing, enabling connections between devices, mobile applications, and browsers to backend services [38].

2.2.2 Communication

As mentioned at the start of this section, Gorums introduces the concept of configurations, allowing developers to invoke RPCs on groups of servers that share a common RPC interface. Multiple configurations, representing any subset of servers in a cluster, can be created and managed by a *Gorums manager*. This struct stores the underlying data structures related to the connections and gRPC streams created between a particular client and the servers in the configurations. Creating several configurations with overlapping sets of servers will not create duplicate sets of data structures and connections.

Regarding the underlying data structures of a configuration, each server in a configuration has a corresponding node struct and a persistent, reliable channel struct, which includes retry and back-off mechanisms in case of failures. Gorums creates a gRPC stream for communication, even for single RPCs between a client and a server, which is managed by the channel. The channel tries to keep the stream open and alive for the entire lifetime of the nodes. To achieve this, the channels are equipped with reconnection capabilities that leverage a backoff configuration. The default configuration used is the one provided by gRPC [41].

The gRPC framework uses protocol buffers [42] for serializing messages in RPCs. These messages are defined in separate protocol buffer files by the developer and compiled into Go code [43]. Gorums has a custom *protoc* compiler that generates its specific functionality, including wrapping the original proto messages with additional metadata. This metadata contains the message ID, the gRPC method being called, and a gRPC error status field, and is used to route messages to the correct handlers. This metadata is extended to enable the additional functionality presented later in this thesis.

2.2.3 Quorum Call

A QuorumCall is a method on the Gorums configuration struct responsible for sending requests to all the servers in the configuration and collecting the responses. This provides a unified interface for RPCs on a group of servers. Every method with the QuorumCall option provided in the proto specification will be generated as a method on the Gorums configuration with the same signature [16][9].

2.2.4 Quorum Function

In a QuorumCall, an identical request is transmitted to all the servers in the configuration but it can accept different responses from the servers. However, the QuorumCall only returns a single response masking the fact that it has received multiple responses [16]. Hence, the user must implement a Quorum Function to collect these responses which determines when and what to return. Fig. 2.3

visualizes how Gorums transparently invokes an RPC with the same request on all servers in configuration and collects different responses into a single response which is returned by the QuorumCall.

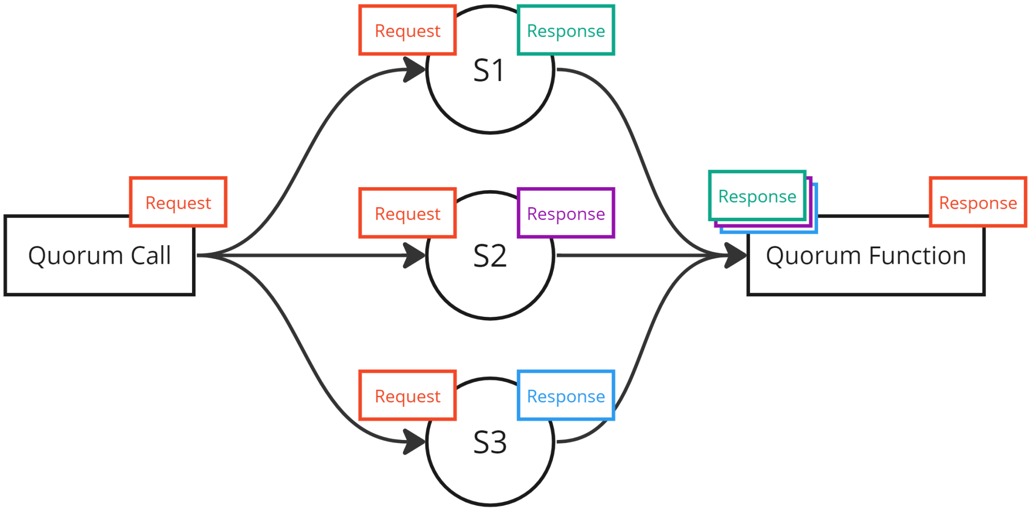


Figure 2.3: A QuorumCall invokes an RPC on all the servers in the configuration with an identical request. Each server may return a unique response, represented in different colors. All responses are processed by the quorum function which unifies them and decides when and what to return.

The user of Gorums must implement a quorum function for each RPC method on the Gorums configuration. Based on the RPCs in the proto specification, Gorums will generate an interface named QuorumSpec containing all the methods. An example is shown in Algorithm 1, where only a single method named "Method" is defined on the configuration. The user is responsible for creating a struct that implements the QuorumSpec interface.

Algorithm 1 Example QuorumSpec Interface

```
type QuorumSpec interface {
    gorums.ConfigOption
    MethodQF(in *Request, replies map[uint32]*Response) (*Response, bool)
}
```

The concrete implementation of QuorumSpec is responsible for collecting the responses from all servers specified in the configuration. Algorithm 2 shows an example implementation of a quorum function that implements the interface shown in Algorithm 1. Note the naming of the method on the QSpec struct: MethodQF. Each method in the QuorumSpec will have the letters QF, abbreviated from *Quorum Function*, appended to the original method name. Furthermore, the quorum function takes the request and a slice of replies as arguments and returns a single response and a bool. This method is run once for each reply from the servers, and the execution is considered complete when the function returns a response and true. The quorum function is quite flexible because the implementer can decide what the quorum size is and how to aggregate the responses. In this example, only one quorum size is used, but the implementer can freely choose to create and use several quorum sizes, such as in Copilot [44, p. 587-588], within the same quorum function.

Algorithm 2 Example Quorum Function

```
func (qs QSpec) MethodQF(in *Request, replies map[uint32]*Response) (*Response, bool)
    if len(replies) $<$ qs.quorumSize then
        return nil, false
    for reply := range replies do
        return reply, true                              ▷ Return the first reply in replies
```

A QuorumCall can be invoked on the configuration as shown in Eq. (2.1), and the reply will be the same as the reply returned from the quorum function. The error returned is generated by Gorums and is determined based on how many replies have been received. If enough servers are either faulty or have returned an error such that a quorum of replies is never reached, an *IncompleteError* will be returned. Otherwise, the error will be nil [16].

$$reply,\ err := \textit{configuration}.\text{Method}(\textit{args}) \tag{2.1}$$

2.3 Practical Byzantine Fault Tolerance (PBFT)

The Practical Byzantine Fault Tolerance (PBFT) is a replication algorithm designed to tolerate Byzantine faults [20]. It ensures both liveness and safety, provided that at most f out of $3f + 1$ replicas are faulty. The algorithm guarantees safety in asynchronous systems and does not rely on any synchrony assumptions.

PBFT operates as a state machine replicated across multiple nodes, where each replica maintains the service state and executes service operations. Clients send requests to the replicas, and PBFT ensures that all non-faulty replicas execute these operations in the same order. Since the replicas are deterministic and start from the same state, all non-faulty replicas produce identical results for each operation. The client waits for $f + 1$ replies from different replicas with the same result. Since at least one of these replicas is non-faulty, this result is considered correct [45, p. 404-405].

The PBFT algorithm consists of three phases: *pre-prepare*, *prepare*, and *commit*, as illustrated in Fig. 2.4. In each view, one of the replicas acts as the leader, known as the primary, while the other nodes serve as backups. The pre-prepare and prepare phases ensure total ordering of requests within the same view, even if the primary is faulty. The prepare and commit phases guarantee that committed requests are totally ordered across different views [45, p. 405-407].

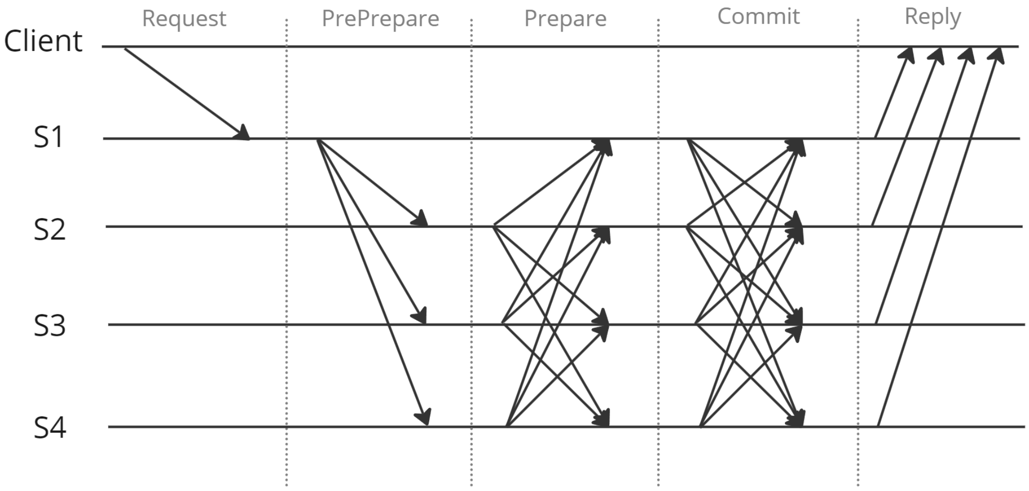


Figure 2.4: Common case operation: replica S1 (primary) assigns sequence number n to request m in its current view v and multicasts a *preprepare* message to the other replicas (backups). If a backup replica agrees, it multicasts a matching *prepare* message. When a quorum of *prepares* is received at a replica, it sends a *commit* message. Replicas execute m after receiving *commit* messages from a quorum, followed by sending a reply to the client [45, p. 408].

Each replica's state includes the service state, a message log of accepted or sent messages, and an integer indicating the current view of the replica. This state can be maintained in volatile memory and does not require stability [45, p. 407]. PBFT ensures liveness even if the primary fails by enabling the system to install new views. It also employs symmetric cryptography to authenticate messages [45, p. 410].

2.4 Paxos

Paxos [30] is a well-known consensus protocol used to implement fault-tolerant distributed systems. Paxos involves three agents: proposers, acceptors, and learners, allowing a single process to act as more than one role. It is assumed that these roles can communicate with each other by sending messages. Proposers are responsible for proposing new values to acceptors, who are responsible for accepting these proposed values. A value is said to be chosen if the acceptors reach an agreement on a proposal. Learners then learn the chosen values. To ensure progress, Paxos designates a leader to act as the proposer. The Paxos protocol is illustrated in Fig. 2.5 and is divided into two phases, each with two steps [30, p. 5-6]:

Phase 1a) A proposer chooses a proposal number n and sends a $prepare$ message with this number to a majority of the acceptors.

Phase 1b) When an acceptor receives a $prepare$ message with a number n that is higher than any previous $prepare$ messages it has responded to, it replies with a $promise$ not to accept any proposals numbered less than n. It also includes the highest-numbered proposal it has accepted, if any.

Phase 2a) Upon receiving responses to its $prepare$ message from a majority of acceptors, the proposer sends an $accept$ message to those acceptors for a proposal numbered n with a value v. The value v is either the value of the highest-numbered proposal among the responses or any value if no proposals were reported.

Phase 2b) An acceptor receiving an $accept$ message for a proposal numbered n will accept the proposal unless it has already responded to a $prepare$ message with a number greater than n. When accepting a value, the acceptors broadcast a $learn$ message to the learners. Once a majority of $learn$ messages are received, the value is considered chosen.

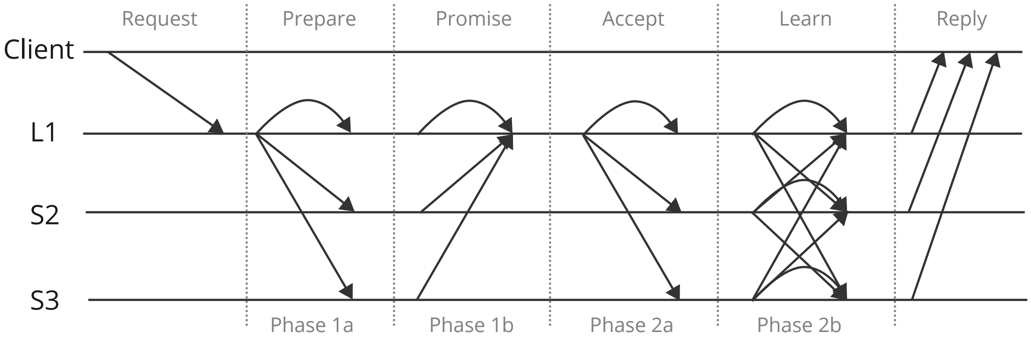


Figure 2.5: Process-centric version of Paxos: Upon receiving a request from a client, the leader sends a prepare message to the other replicas, which respond with a promise. The leader then sends an accept message with the proposed value. The replicas then broadcast a learn message before replying to the client [30].

T. C. Frausing adapted a data-centric version of Paxos in his master's thesis [46] and implemented it in Gorums, inspired by Disk Paxos [47] and Active Disk Paxos [48]. Fig. 2.6 shows the adaptation of the Paxos protocol into a data-centric version, in contrast to the process-centric version depicted in Fig. 2.5, which uses server-to-server communication. Because the learners do not receive a quorum of learns, a third phase is introduced: *Commit*. In this phase, the leader broadcasts a commit message with the chosen value to all learners after receiving a majority of learns from the acceptors. This version is based on the approach where the acceptors respond with their acceptances to a distinguished leader, as described in [30, p. 6].

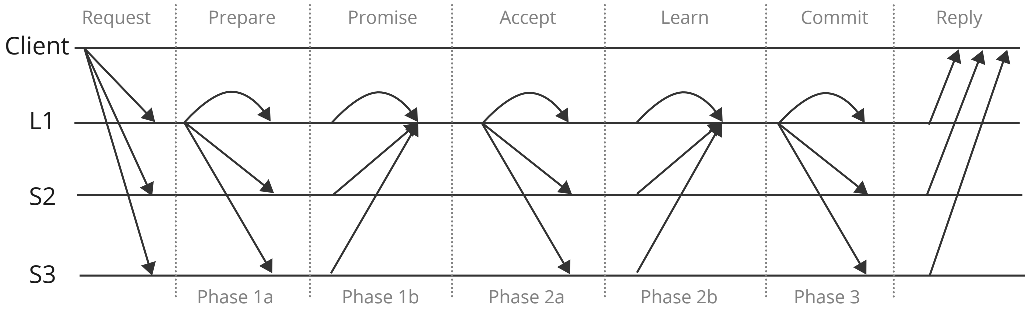


Figure 2.6: Data-centric version of Paxos: Upon receiving a request, the leader sends a prepare message to the other replicas, which respond with a promise. The leader then sends an accept message with the proposed value. The replicas respond to the leader with a learn message. Finally, the leader broadcasts a commit message before the replicas respond to the client [46].

Chapter 3

Related Work

This thesis builds upon previous work presented in [16] and [9], extending existing functionality to accommodate all-to-all communication in Gorums while ensuring compatibility with its current features.

Various systems aim to mitigate challenges in distributed systems, often providing high-level abstractions for application development. As discussed in Section 1.1, these systems are offered as services for user applications. In contrast, this thesis focuses on creating an abstraction for all-to-all communication in quorum-based systems, making these systems potential users of the developed framework.

MapReduce [49] and scatter-gather [50] involve distributing requests to multiple targets, either as tasks or messages, and then aggregating the results. This approach is similar to how a QuorumCall in Gorums operates. However, MapReduce and scatter-gather are designed for general-purpose data processing rather than quorum-based consensus protocols.

The actor model is another approach that aids in building distributed systems by providing powerful abstractions to address some of their challenges [51][52]. It supports asynchronous message passing and enforces encapsulation without using locks [53]. Various implementations of the actor model exist in different languages, for example Erlang [54], Scala [52], and Golang [55]. However, these implementations do not support a unified abstraction for invocations that enable all-to-all communication patterns and relate multiple sub-requests to a single client request.

Open MPI [56], an open-source Message Passing Interface, is primarily used in high-performance computing (HPC) applications where efficient communication between parallel processes is essential. It offers the collective operation *MPI_Alltoall*, where all processes exchange data equally [57]. While Open MPI can support high-performance quorum-based systems, it requires careful design as it is geared towards HPC. Conversely, Gorums and the framework developed in this thesis are specifically designed for fault-tolerant quorum-based systems, aiming to simplify their development.

Dependably Fast (DepFast) [13] is an expressive framework for developing quorum systems, introducing a QuorumEvent abstraction for building such systems in a synchronous style. DepFast facilitates the implementation of complex quorum systems with high performance, using coroutines, whereas Gorums utilizes goroutines. The QuorumEvent abstraction in DepFast allows developers to write synchronous-style code, similar to the QuorumCall and quorum function in Gorums. However, unlike the quorum function, which processes individual messages separately from the invocation and returns only a single reply, QuorumEvent returns messages based on a waiting criterion without first processing them. The framework presented in this thesis extends the QuorumCall functionality by enabling all-to-all communication within a single invocation, a feature not directly supported by DepFast.

Chapter 4

Design

This chapter first highlights different requirements and challenges related to creating a quorum-based framework that enables all-to-all communication. These create the foundation for what features the framework offers and how it is designed. For the rest of this thesis, the new framework is named *broadcast framework* to distinguish the new functionality from the existing already provided by Gorums. In the last part of the chapter, we will present the system specification and architecture of the proposed broadcast framework.

4.1 Analysis

In this section, we will present various issues and design requirements related to building a framework that enables the all-to-all communication pattern.

4.1.1 Practical Systems

A range of algorithms and systems using all-to-all communication to some extent have been analyzed. A goal in this study has been to make the broadcast framework as generic as possible, hence accommodating most quorum-based algorithms. The algorithms analyzed will be presented as a bullet list in short here. The goal is to set a common interface for all algorithms and only focus on how the all-to-all communication is used. We thus try to limit the surface when explaining the design decisions further in this chapter. I.e. we will present what the algorithms share in common and only use PBFT and Paxos when explaining design choices later. Only when an algorithm has unique specifications will the ideas be presented at later points.

- *Gossip protocols* (e.g. Eager Probabilistic Broadcast [11, p. 141]): Requires being able to broadcast to a subset of nodes with an arbitrary amount of steps.

- *Broadcasting algorithms* (e.g. FIFO and Casual Broadcast [11, p. 149]): Allows nodes to disseminate information from one or more nodes to all other nodes. Furthermore, All-Ack Uniform Reliable Broadcast allows for delivering a message outside of the function receiving the broadcast messages [11, p. 127].

- *Paxos*: Can use the all-to-all pattern when sending learn messages as explained in Section 2.4. Replicas include themselves in the broadcast.

- *PBFT*: Uses the all-to-all pattern in the prepare and commit phases. The primary and the backups do not include themselves in the broadcast. Also, the primary does not send prepare messages meaning the sequence of execution is different on the primary and the backups.

- *Tendermint*: Uses the all-to-all communication pattern during the block proposal and validation stages. The proposer initially broadcasts the block proposal to all validators, who then broadcast in three phases: pre-vote, pre-commit, and commit. They wait for 2/3 of the messages in each phase before progressing. The block is finalized in the commit phase [58]. Nodes only broadcast to each other and do not follow a traditional request-response pattern.

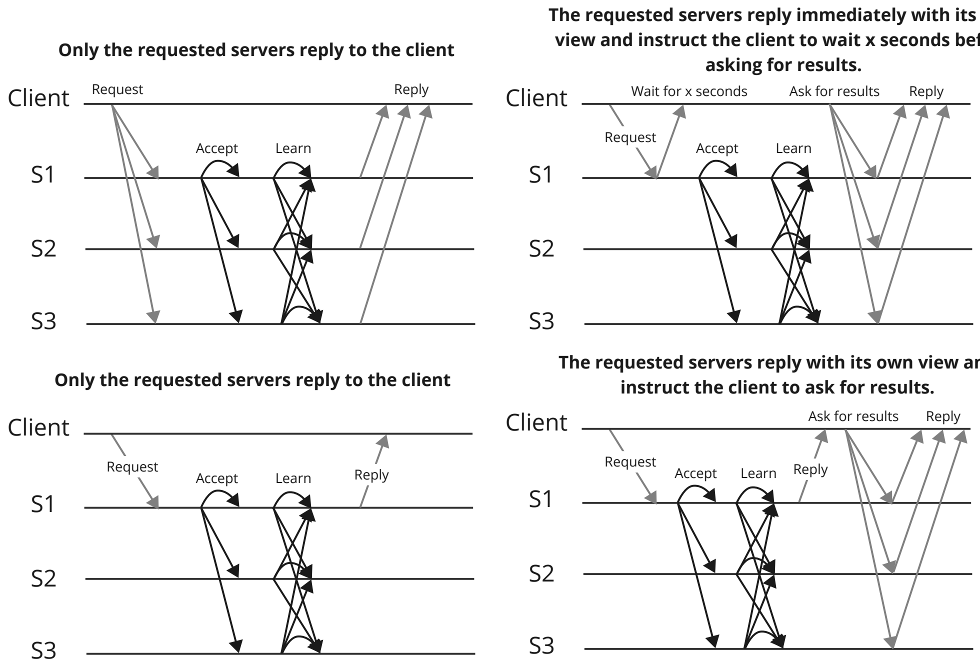
- *Stellar Consensus Protocol*: This is a construction of the Federated Byzantine Agreement with multiple rounds of all-to-all communication among quorum slices [59]. This protocol does not directly use all-to-all communication where every node communicates directly to all other nodes. Instead, nodes communicate with a subset called quorum slices, which further interact with their slices. Each step involves multiple nodes communicating without being directly connected with the initiating node.

4.1.2 Gorums

The broadcast framework extends the current functionality of Gorums, meaning most of the underlying infrastructure is used such as configurations, nodes, channels, and encoding (serialization/deserialization). Gorums provide reconnection and back-off strategies making connections reliable. However, Gorums is currently limited to a request-response pattern that does not allow for requests to be involved in several execution steps involving different handlers and servers. Therefore, the functionality of Gorums needs to be extended to address these limitations.

4.1.3 Communication

Several design alternatives exist for client-server cluster communication, illustrated in Fig. 4.1 and Fig. 4.2. One approach is for the client to broadcast to all servers and collect a quorum of responses shown in the first scenario in Fig. 4.1. Alternatively, the client can communicate solely with the cluster leader, receiving responses only from the leader as illustrated in the bottom left scenario in the figure. Another pattern involves the client sending a broadcast request to the leader and then collecting results from all servers, either after a predefined time t or upon receiving the leader's response. All methods eliminate the need for client-side response handlers. Furthermore, the client can send requests to a subset of servers, as shown in Fig. 4.2, but requires a handler to collect responses from all servers. The client can initially receive responses from the requested servers, as depicted in the second scenario in the figure, and subsequently receive responses from all servers. The first scenario in Fig. 4.2 closely resembles PBFT and is, therefore, the preferred communication pattern. This also implicitly supports the first and third communication patterns shown in Fig. 4.1. This design requires the client to run a server, which imposes overhead on the client side and blurs the separation between a client and a server. Therefore, the client-server should be as minimal as possible.



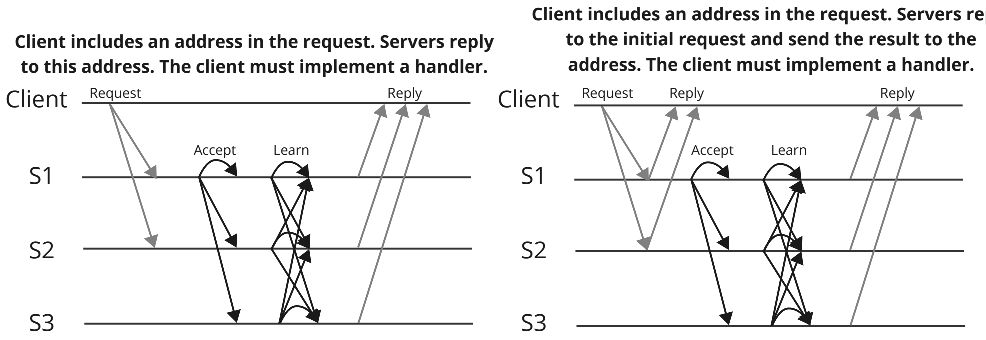
Only the requested servers reply to the client

The requested servers reply immediately with its own view and instruct the client to wait x seconds before asking for results.

Only the requested servers reply to the client

The requested servers reply with its own view and instruct the client to ask for results.

Figure 4.1: Communication alternatives that do not require implementing a client-side handler to collect responses.



Client includes an address in the request. Servers reply to this address. The client must implement a handler.

Client includes an address in the request. Servers reply to the initial request and send the result to the address. The client must implement a handler.

Figure 4.2: Communication alternatives that do require implementing a client-side handler to collect responses.

In Tendermint, no client is issuing a request but instead, a proposer initiates a request handled by validators, as detailed in Section 4.1.1. The validator set, which includes the proposer and the validators, can be considered as servers. These servers communicate exclusively through broadcasting. Fig. 4.3 illustrates a scenario where a server initiates a request, and each server concludes the request after a broadcast step without replying to the initiating server.

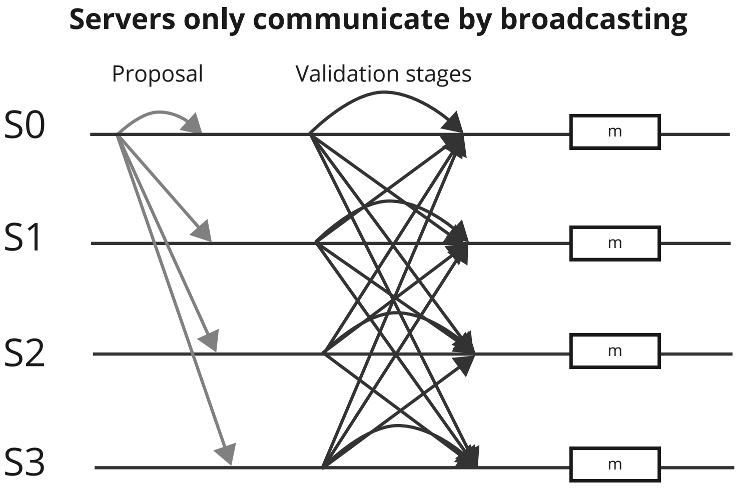


Figure 4.3: Communication alternatives, special case where servers only communicate by broadcasting to each other.

4.1.4 Request

A request needs to encompass a broader scope and should no longer be confined to a single request-response between a client-server or server-server pair. The lifespan of a request should start when the client initiates it and end when the client receives a response from the servers. Behind the scenes, the request may involve several sub-requests spanning different handlers and servers. For example, a write request sent to replicas running Paxos could be initiated by a client, starting a round of Paxos that concludes only when the round has ended and a response is received by the client. In this case, the messages sent between the servers during the prepare, accept, and learn phases would all be related to the original client request.

Defining the lifetime of such client requests is essential to prevent issues related to stale or old data. Client requests should have a short lifespan and be garbage collected eventually, without interfering with the protocols running. Removing client requests too early could potentially violate safety properties in protocols. For instance, in PBFT, if removing a client request marks the end of the execution, it could prevent a backup from broadcasting. Consequently, a correct backup could end up committing a value without broadcasting this to the other backups due to the premature removal of the client request. Therefore, a client request should be considered handled if it has been successfully executed or canceled. Successful execution occurs when the server sends a response to the client, while cancellation depends on the user's implementation. After the client request has been successfully terminated, data related to it should be properly garbage collected on each server.

4.1.5 Broadcasting Context

Requests can be added to a queue to be processed later by the servers, meaning they are temporarily stored and transmitted upon meeting certain criteria. For example, in Paxos, when the leader sends a prepare message, the replicas reply with a promise containing accepted messages. These accepted messages once belonged to client requests. Since there is no guarantee that the leader has seen these messages, it must be able to send an accept for all missing slots. The leader is a server, so it must be possible to broadcast messages outside the server handler context.

4.1.6 Cancellations

A timeout specifies a set number of seconds during which the client is unwilling to wait for a response from a server. This is a simple yet important concept in building robust distributed systems, allowing clients to avoid unnecessary waiting and enabling servers to stop processing, thereby improving resource utilization and system latency [60]. Extending this idea, systems can be designed to send cancellations for operations, improving performance by reducing the amount of work needed [61]. However, implementing cancellations is not trivial and requires careful design. Some protocol specifications, such as Paxos and PBFT, do not inherently support the cancellation of requests, as these can occur at any time during execution. Cancellations can be intentional or caused by factors such as server or client failures, impatient clients, malicious actors, or slow networks.

We will highlight different scenarios where cancellations could interfere with protocol correctness, using PBFT as an example. Although PBFT inherently addresses many issues, we will make some assumptions to explore the impact of cancellations: canceling a request means servers stop accepting new messages and revert the state related to the client request, treating it as if it were never sent. Additionally, clients act as the source of truth, meaning a client request is only considered failed if it actually fails. To simplify the examples, we will use three servers, meaning none can fail according to PBFT. A fourth server will be introduced only when illustrating a failure scenario. Our goal is to demonstrate whether cancellations can be supported by the framework by verifying the presented strategies against the correctness of PBFT. We start with the two assumptions illustrated in Fig. 4.4:

1. A client can cancel a request: This is done by either disconnecting or canceling the context provided with the request. In both cases, $\leftarrow ctx.\text{Done}()$ returns, causing the servers to propagate the cancellation to the other servers.

2. Servers are configured with a deadline, after which they cancel the request internally but do not send cancellations to other servers.

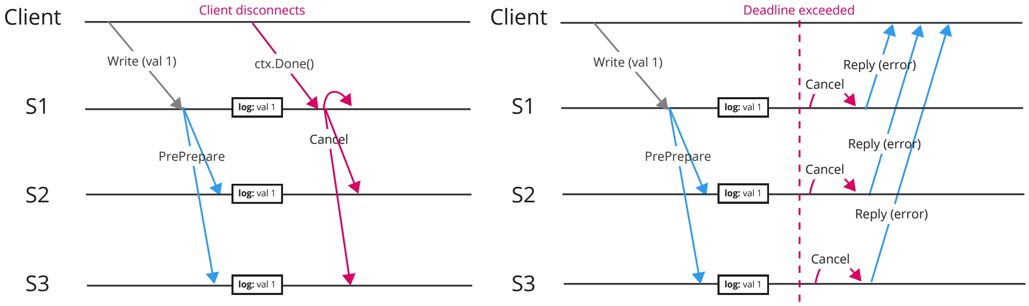


Figure 4.4: *Case 1:* A client can cancel a request by either disconnecting or canceling the context. *Case 2:* Servers can cancel a request internally after a given deadline.

Fig. 4.5, Fig. 4.6, and Fig. 4.7 illustrate three examples, respectively, where this approach could cause ambiguity and either lead to safety violations or impede progress:

1. **Deadline**: The first example shows a scenario where the deadline is set too short, causing S3 to time out before receiving the last commits. Although the execution is correct according to PBFT, S3 is prevented from making progress due to the deadline.

2. **Server Failure**: In the second example, the client knows only about S1 and sends the request to S1, but then S1 crashes. Should the client regard this as

a failed request, or should it wait in case it receives replies from other servers it does not know about? If it considers the request failed immediately when S1 crashes, it will not know about a successful execution without checking later. If it waits, it could be stuck indefinitely without using a timeout.

3. **Client and Server Failure**: The third example shows a scenario where both S1 and the client crash. Since S1 crashes before the client, it cannot propagate the cancellation. According to our assumptions, S2, S3, and S4 should have canceled the request and reverted the state.

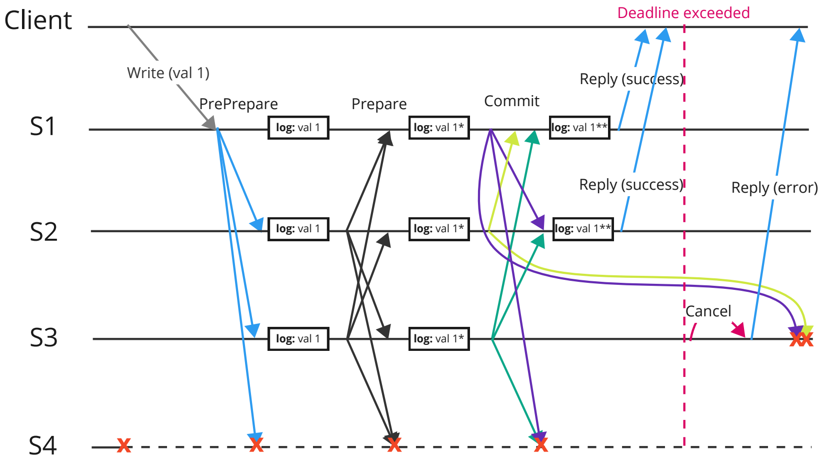


Figure 4.5: *Deadline*: S3 cancels the request even though the execution was correct.

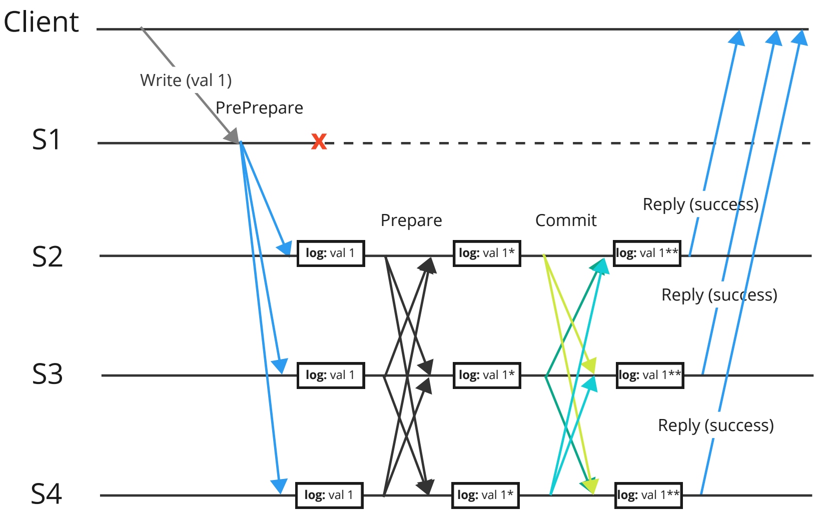


Figure 4.6: *Server Failure*: S1 fails and the client does not know about the other servers.

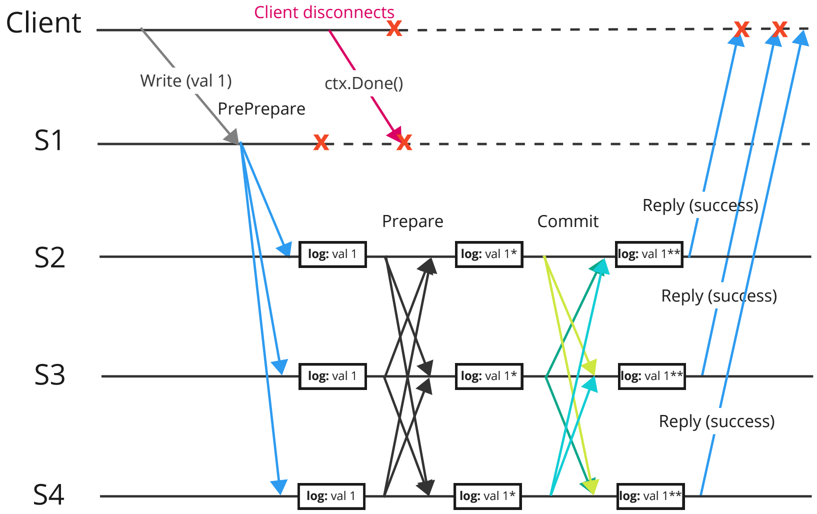


Figure 4.7: *Client and Server Failure*: The client fails after S1 without the other servers knowing.

To address the issues mentioned above, we could instead say that servers should broadcast cancellations when the deadline is exceeded. We will require that servers invoke the last method in response to receiving cancellations. However, if a server receives a quorum of cancellations, it will revert the state and broadcast a cancellation. Additionally, we assume that the client uses a timeout for requests but does not send cancellations when the deadline is exceeded. This scenario is displayed in Fig. 4.8 and Fig. 4.9:

1. **Deadline**: S1 and S2 respond with the last value, correctly executing the commit and consequently replying to the client. When S3 broadcasts a cancellation, both S1 and S2 reply with the commits that previously failed, bringing S3 to the correct state. However, if the client uses a timeout that is too short, it would still regard the request as failed because it did not receive enough replies within the deadline.

2. **Performance**: S1 successfully executes the request, but the other servers fail to do so. Hence, a majority of cancellations would be broadcast, and all servers would revert their state. This could cause ambiguity for the client, which received a successful reply from the only server it sent the request to, S1. It might be worthwhile for the client to reset the timeout and wait for more replies, but this could lead to poor performance.

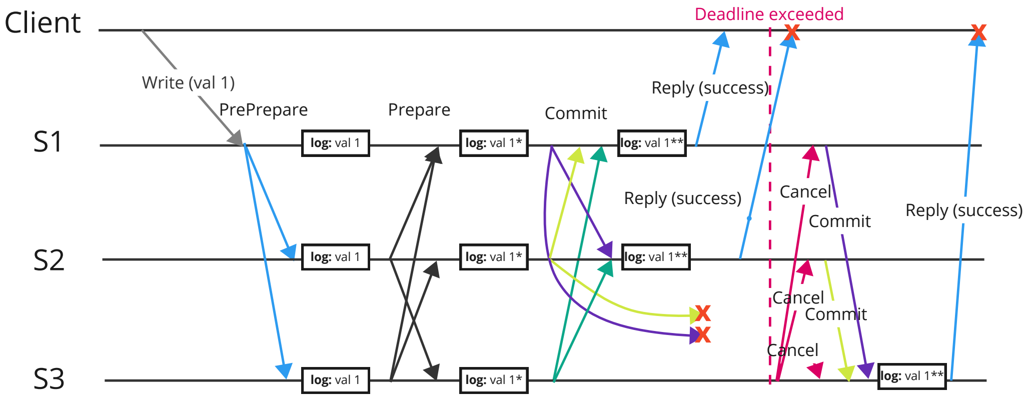


Figure 4.8: *Deadline*: The servers manage to arrive at a correct state after S3 broadcasts cancellations, but the client has timed out.

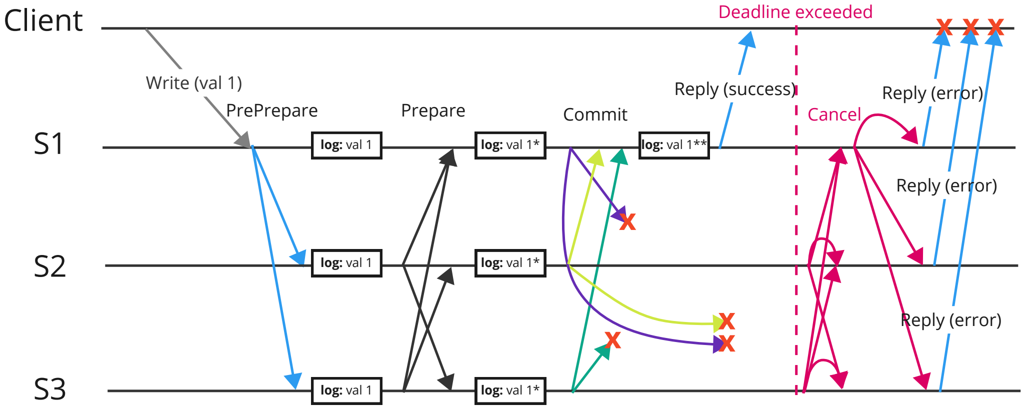


Figure 4.9: *Performance*: The client receives a reply from S1, but the servers agree on canceling the request.

Building on the previous scenarios, we could allow the client to broadcast cancellations when the deadline is exceeded. The client would send cancellations to all servers it knows about and to all servers that have sent a reply. This is illustrated in Fig. 4.10 and Fig. 4.11. Additionally, servers would not broadcast cancellations when their deadlines are exceeded. Based on the figures, we have these cases:

1. **Cancellation**: The client successfully cancels a request where all servers revert the state.
2. **Inconsistency**: If the client fails after S2 has sent a response, S2 will not revert its state. This is because the client only sent the request to S1, and therefore, S2 will not know whether the client is still running or not.

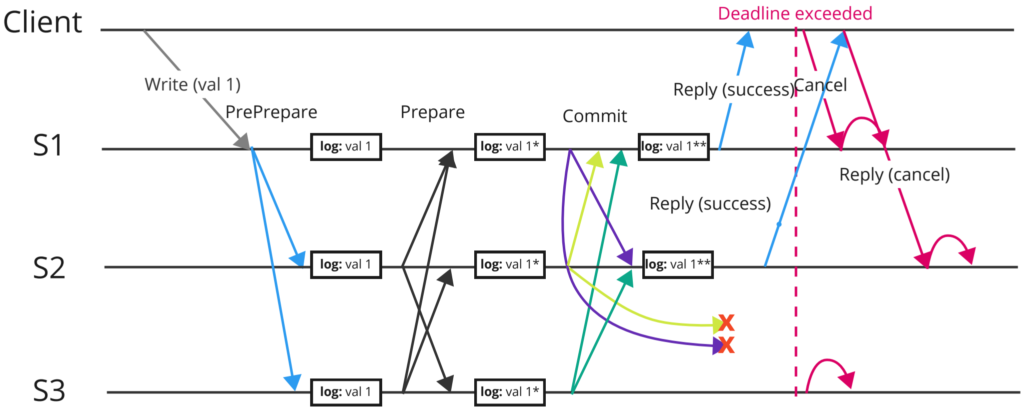


Figure 4.10: *Cancellation*: The client successfully cancels the request.

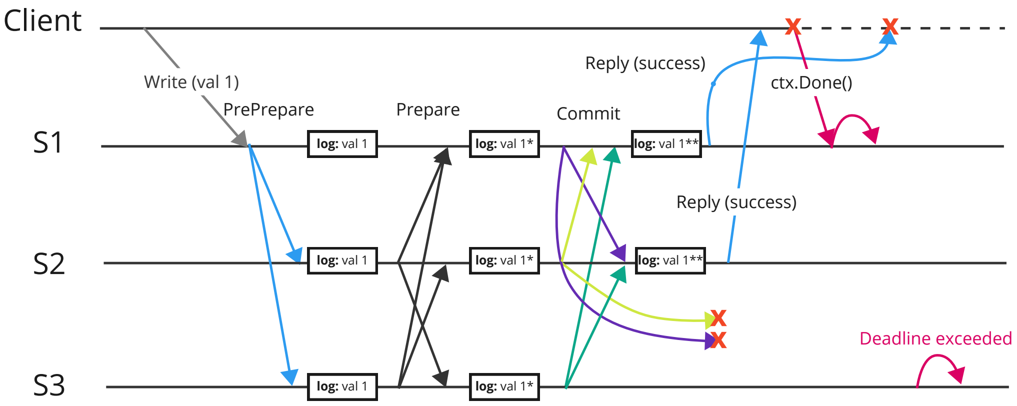


Figure 4.11: *Inconsistency*: The state is inconsistent across the servers due to S2 not knowing about the client failure.

Another solution could be for the client to send acknowledgments to successful replies from servers, as shown in Fig. 4.12 and Fig. 4.13. Furthermore, servers would cancel a request if they have not received an acknowledgment from the client within the deadline. However, this approach is susceptible to the same issue where the client fails after S2 has sent a successful reply, as shown in Fig. 4.11. Additionally, it can be problematic, as illustrated in Fig. 4.13, where the reply from S3 fails in transit and the request is canceled. The other two servers receive an acknowledgment within the deadline, assuming the request is successful, and thus the state across the servers becomes inconsistent.

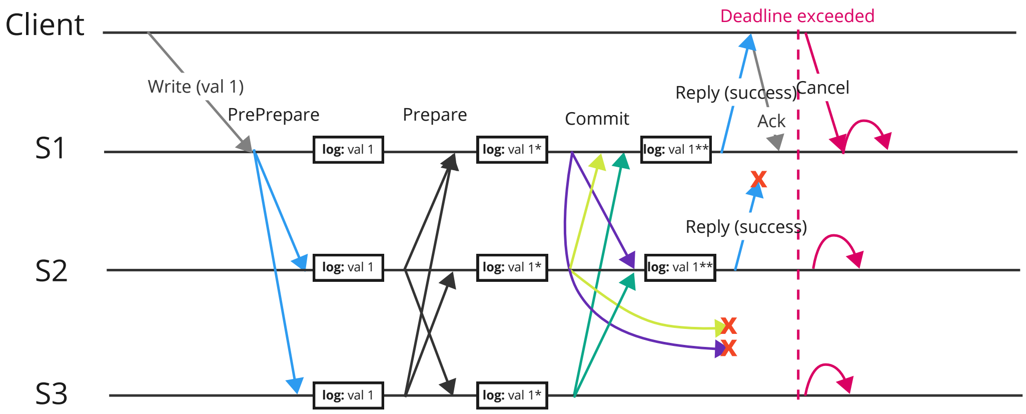


Figure 4.12: *Cancellation*: Servers cancel the request if they have not received acknowledgements within the deadline.

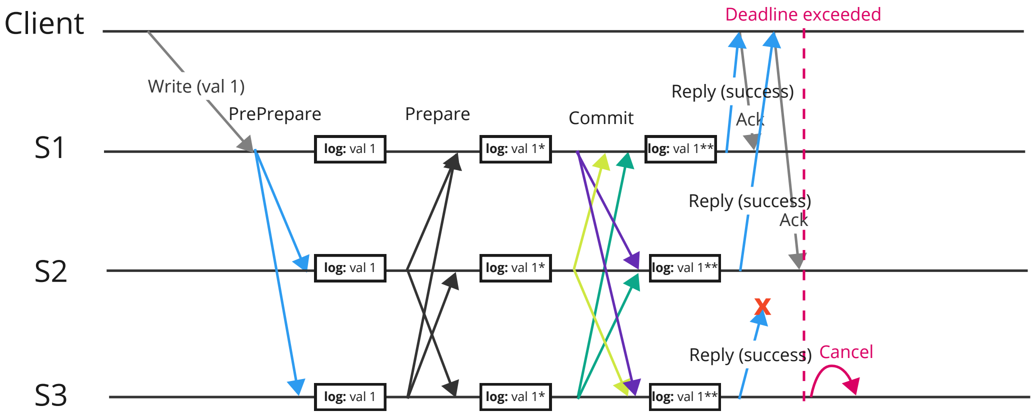


Figure 4.13: *Inconsistency*: S3 does not receive an acknowledgement within the deadline, while the other servers do.

In a final effort to propose a cancellation mechanism that could work for PBFT, we provide the scenario in Fig. 4.14 and Fig. 4.15. Only clients are allowed to send cancellations, and this happens when the deadline is exceeded or if the client fails. The client will then send cancellations to servers that it knows about and to all servers that have sent a reply to it. Servers are not permitted to cancel a request on their own and regard a request as successful if they have sent a reply to the client and the client has not sent a cancellation for the request. We then present the two cases in the figures:

1. **Cancellation**: The request is successfully reverted after the deadline is

exceeded, and all relevant servers receive the cancellation.

2. **Inconsistency**: The reply from S3 is not received by the client, and S3 regards the request as successful because it does not receive a cancellation. This leads to an inconsistent state since S3 believes the request succeeded while the other servers have reverted it.

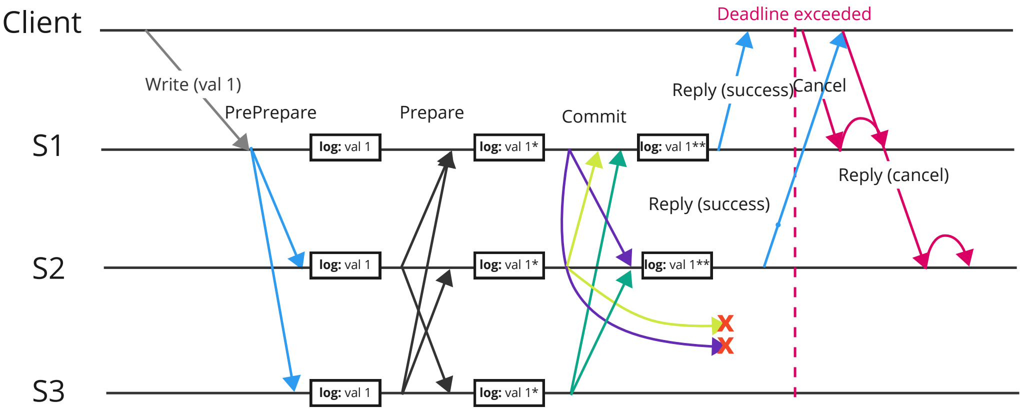


Figure 4.14: *Cancellation*: The request is successfully canceled.

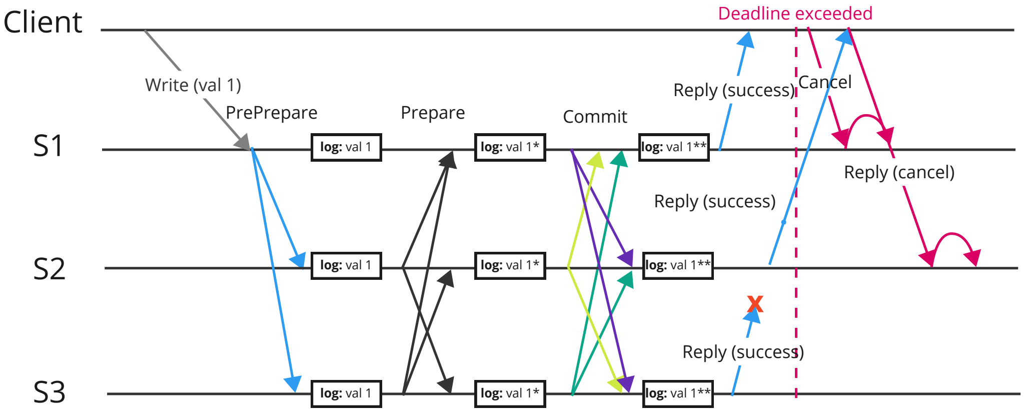


Figure 4.15: *Inconsistency*: S3 failed to send the reply but regarded the request as successful because it did not receive a cancellation from the client.

In any case, a cleanup function would need to be implemented by the user, diminishing much of the benefit of using a framework for cancellations. We will therefore conclude by offering only a weak cancellation guarantee. A client should be able to send cancellations to servers, but it is up to the user and each particular protocol to decide whether or not to handle the cancellations. The framework should provide functionality for propagating cancellations and listening for such messages, but it should not enforce their use.

4.1.7 Ordering

The order in which messages arrive is a well-known issue in distributed systems, with several protocols proposed to address these problems, such as FIFO and Causal Broadcast [11, p. 149]. However, enforcing these algorithms can create significant overhead if message ordering is not required. A simple remedy to provide some level of ordering could be to constrain each server or client to guarantee that they only send messages in order. This could be achieved by using an internal queue and only proceeding to send the next message if the previous one was successful. However, as illustrated in Fig. 4.16 and Fig. 4.17, this method would not guarantee ordering in scenarios where all-to-all communication is allowed. In the first figure, S1 is enforced to send messages in order, so the learn message is always sent after the accept message from S1. However, S2 is allowed to send the learn message before S3 receives the accept message from S1. In Fig. 4.16, a similar problem occurs if the client fails to send m1 but the other servers manage to do so.

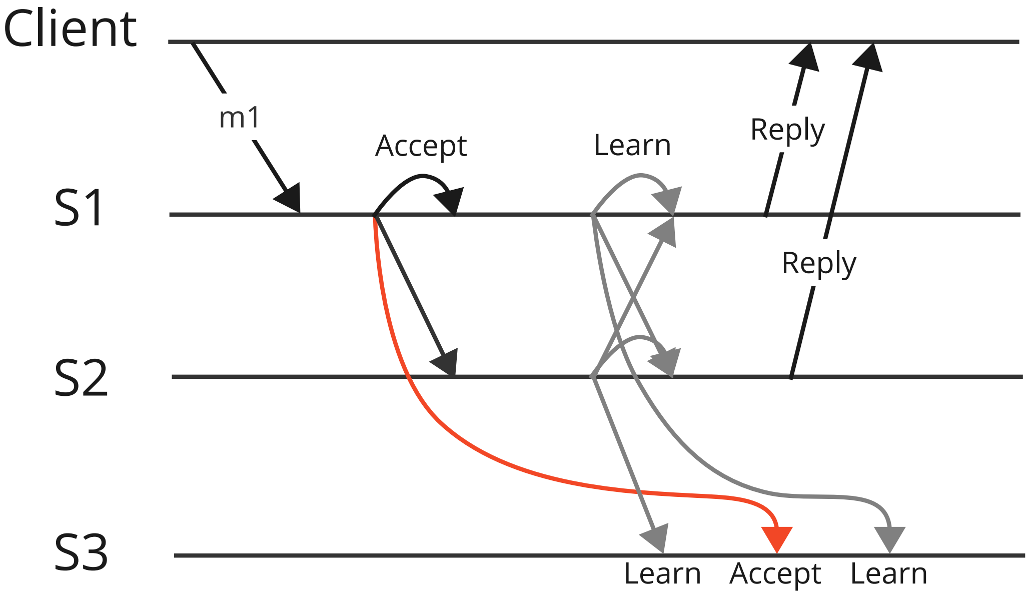


Figure 4.16: Example of messages arriving out of order: Servers are only allowed to send messages in order. If the accept message arrives late to S3 from S1, then S2 could manage to send a learn message before the accept message arrives.

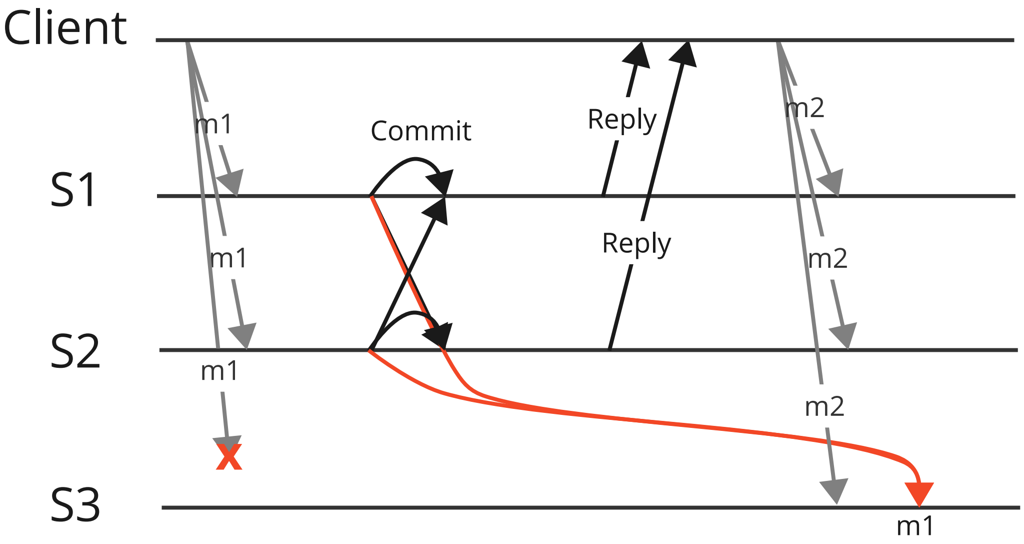


Figure 4.17: Example of messages arriving out of order: The client is only allowed to send messages to the servers. If m1 fails to be sent from the client, S3 can receive m2 before m1.

An ordering-related problem can also arise in scenarios where it is not directly expected. For example, PBFT specifies that messages may arrive out of order [20]. However, poor implementation of the protocol can lead to scenarios where the servers do not make progress. Fig. 4.18 illustrates an ordering issue related to PBFT. If all commit messages arrive before the prepare messages, the primary would not evaluate the state as being prepared when receiving any of the commit messages. When it eventually receives the prepare messages, it will need to run the commit logic again. This means that the commit messages should have been cached when they first arrived. The order in which handlers are executed, or simply the execution ordering, is crucial for PBFT and should be managed by the framework.

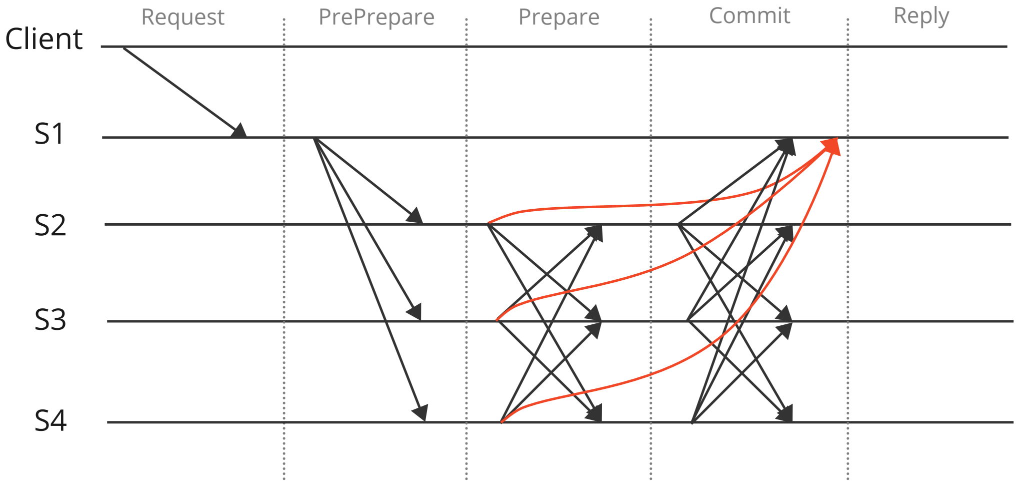


Figure 4.18: Example of messages arriving out of order: If messages are not cached on the servers, then S1 would not make progress due to receiving all commit messages before the prepare messages.

Another ordering issue can occur if a server relies on receiving a client request to respond to it. For example, in Paxos, a replica might finish a round before receiving the original request from the client. This can happen if the client is slow and the request is received by the leader before reaching the other replicas as illustrated in Fig. 4.19. Therefore, the response should be cached until the replica can respond to the client request.

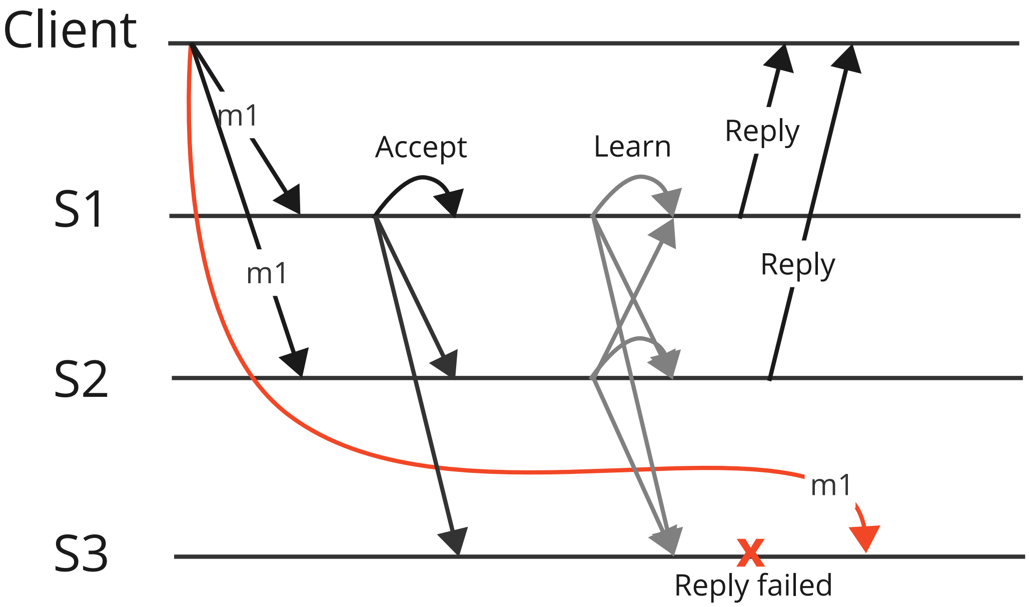


Figure 4.19: An example of messages arriving out of order at one of the replicas in Paxos. In this scenario, replicas require a request from the client to be able to reply. If responses are not cached on the servers, then S3 will fail to send a response because it has not seen the request from the client before trying to respond.

4.1.8 Message Duplication

Broadcasting should be constrained to prevent flooding the network and potentially causing denial of service. The all-to-all communication pattern inherently produces $O(n^2)$ messages. Ideally, each broadcast to a specific method should only happen once per client request. Broadcasting from a server handler can lead to multiple invocations of the broadcast method if many messages are received by that handler. Therefore, the framework should account for this and guarantee that each broadcast only occurs once.

4.1.9 Authentication

In a Byzantine process abstraction, each client request must be validated using cryptographic algorithms. It is important that messages are not only encrypted in transit but that the sender is also authenticated. Since requests can originate from a client and be processed by an intermediary server, the origin must also be properly authenticated. This process induces overhead and should be optional for the user to enable.

4.2 Specification

Based on the analysis performed in the previous section, we can create a module specification inspired by the format used in [11]. Module 3 outlines the events and properties required to develop a framework that enables all-to-all communication. To maintain consistency with the broadcast framework's naming conventions, we introduce the *broadcast request* which essentially corresponds to the client request defined in the previous section. A more thorough explanation of the broadcast request will be provided in Section 5.2.

Module 3 Specification for the all-to-all Framework

Module:

Name: Broadcast Framework, **instance** *bf*.

Events:

Request: $\langle$ *bf*, *SetView* | π $\rangle$: Requests the broadcast framework to install the view representing the set of servers π.

Request: $\langle$ *bf*, *Broadcast* | $\pi,\ message$ $\rangle$: Requests the broadcast framework to execute a broadcast to the set of servers π with the given $message$.

Indication: $\langle$ *bf*, *QuorumFunction* | $response$ $\rangle$: Indicates that the client has processed all received responses with output $response$.

Request: $\langle$ *bf*, *Cancel* | $\pi,\ cancellation$ $\rangle$: Requests the broadcast framework to execute a $cancellation$ to the set of servers π.

Request: $\langle$ *bf*, *SendToClient* | $c,\ response$ $\rangle$: Requests the broadcast framework to execute a unicast to the client c with the given $response$.

Properties:

BF1: *Termination:* A broadcast request B sent from a client c with upper bound t will return a $response$ within t time units.

BF2: *Retries:* If a server s fails to send a message m, then m will be retried later with upper limit x.

BF3: *Execution ordering:* A message m corresponding to handler H_n will be cached until the order $\psi \geq n$ if execution order is enforced.

BF4: *Uniqueness:* If a server s process broadcast request B_1 and a server p process broadcast request B_2 such that $B_1.id = B_2.id$, then $B_1 = B_2$.

BF5: *Relation:* If a message m related to broadcast request B is processed by a server s, then $B \notin \phi$.

BF6: *Independence:* A message m_1 related to broadcast request B_1 and message m_2 related to broadcast request B_2 will be processed asynchronously and separately on a server s.

BF7: *No duplication:* No message is broadcast more than once.

BF8: *Validity:* No message m related to a broadcast request B such that $B.finished = true$ will be processed by a server s.

BF9: *Authenticity:* If a message m related to broadcast request B with origin c and sender p is processed by server s, then B was initiated by c and m previously sent to s by p.

BF10: *Origin:* If a broadcast request B with origin c is processed by server s and c receives a response from s, then c is a client.

BF11: *Communication:* If a broadcast request B with origin s is processed by server p and p does not send a response related to B, then both s and p are servers and only communicate by broadcasting.

BF12: *Weak Cancellation:* If a cancellation $cancel$ is processed by server s, then s is either directly connected to origin o or all intermediary servers have propagated $cancel$.

BF13: *Garbage Collection:* If a broadcast request B is processed by server s, then B should eventually be cleaned up and removed.

4.3 Architecture

Based on the specification introduced in the previous section, we have decided to extend the QuorumCall to accommodate the all-to-all communication pattern. We have thus created a new call named *BroadcastCall*, illustrated in Fig. 4.20. The BroadcastCall is invoked by a client initiating a broadcast request. The broadcast request can be sent to all or a subset of the servers, and the servers support forwarding messages. As illustrated in the figure, servers are allowed to broadcast unique messages to all servers. The BroadcastCall culminates in sending a response back to the client, which processes the responses in a quorum function. This quorum function returns a single response to the BroadcastCall, making it transparent to the client. To facilitate server-to-server communication only, an alternative to BroadcastCall is provided for servers. This will be explained in Section 5.2.1.

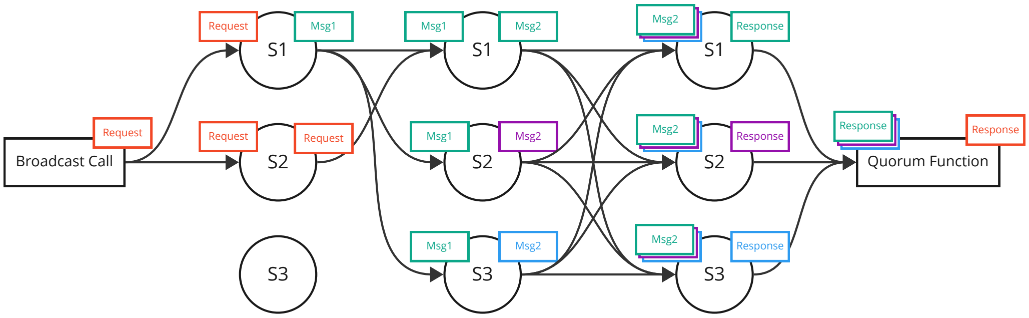


Figure 4.20: BroadcastCall extends the functionality of QuorumCall while maintaining similar semantics. The client sends an identical request to two servers. S1 is the leader and broadcasts a message to the others. S2 is not the leader and forwards the request to the leader. Since the leader has already seen the request, it does nothing. Each server then proceeds to the next stage by processing the message from the leader and broadcasting a new unique message to the others. Finally, the servers process all received messages and send a response back to the client. These responses are then processed by the quorum function.

Fig. 4.21 shows the overall architecture of the newly implemented components that facilitate a BroadcastCall. First, the broadcast request is sent by a client and can comprise several individual messages. Each message is processed separately by the servers through a custom broadcast handler, which hands it off to the broadcast manager, an internal component of each server. The broadcast manager uses shards to store state related to the broadcast requests and enables parallel processing of multiple requests.

For every broadcast request, a broadcast processor is created, which handles the main logic of the broadcast framework. After being processed by the broadcast processor, the message is sent to the user-implemented server handler. Depending on the user implementation, the processor then either broadcasts a new message to the servers or sends a response back to the client, thus completing the broadcast request.

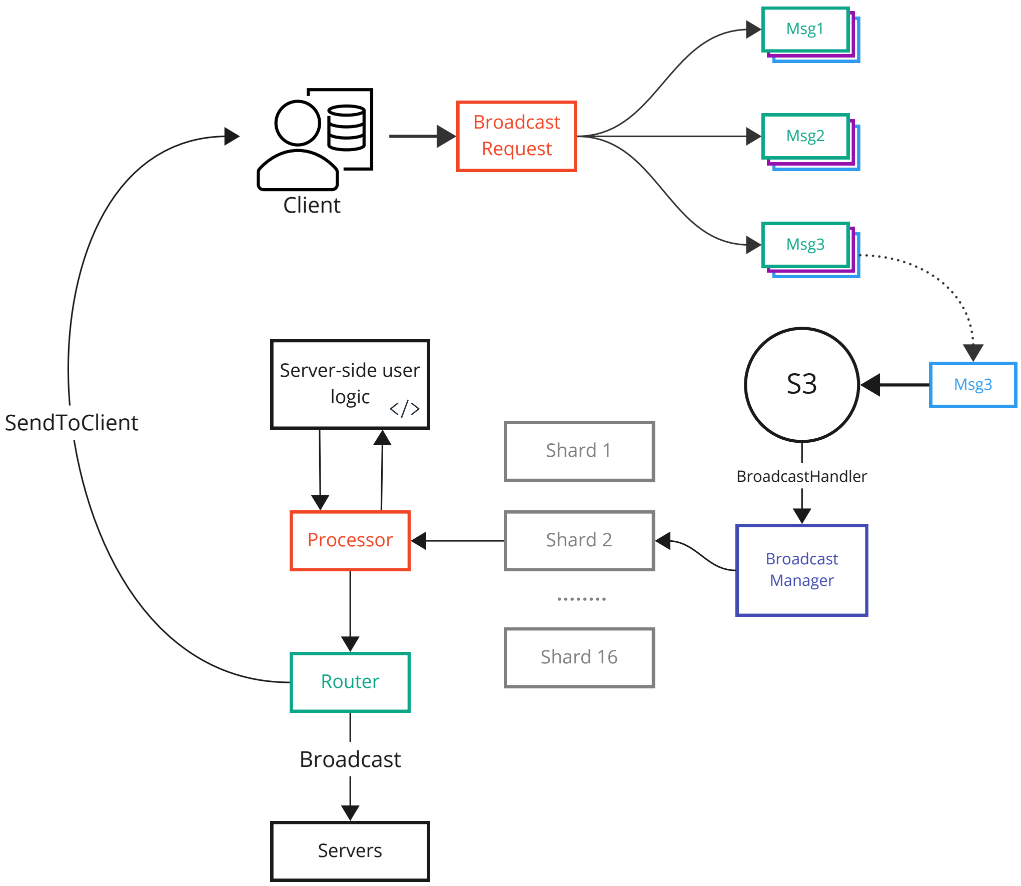


Figure 4.21: Architecture of the broadcast framework.

Chapter 5

Implementation

In this chapter, we will present the implementation of the broadcast framework designed to address the issues discussed in the previous chapter. We will begin by explaining how to enable the broadcast framework features in the proto specification. Following that, we will introduce the broadcast application programming interface (API) before delving into the components of the broadcast architecture described in Section 4.3. Additionally, the user of Gorums is referred to as either the user or the implementer in this chapter.

5.1 Proto Message

The proto specification in Listing 5.1 illustrates how the broadcast functionality can be enabled for a Paxos implementation. Two proto options are provided to fine-tune the behavior of broadcast requests: broadcast and broadcastcall. The broadcast option creates internal handlers for servers and can be used alone or in conjunction with other call options, such as quorumcall. The broadcastcall option, on the other hand, creates a client-side server handler in gRPC, allowing servers in the cluster to send responses to a broadcast request. Methods with the broadcastcall option enabled also act as entry points and are meant to be called only from clients.

For nomenclature purposes, all methods in the proto specification are defined as the set $\mathcal{H}$, and we will use this notation for the remainder of this chapter. The broadcast-enabled handlers, $\mathcal{H}_B$, are defined as the RPC methods with either the broadcastcall option or the broadcast option.

Listing 5.1 : An example proto specification for Paxos.

```
service Paxos {
  rpc Write(PaxosValue) returns (PaxosResponse) {
    option (gorums.broadcastcall) = true;
  }

  rpc Prepare(PrepareMsg) returns (PromiseMsg) {
    option (gorums.quorumcall) = true;
  }

  rpc Accept(AcceptMsg) returns (Empty) {
    option (gorums.broadcast) = true;
  }

  rpc Learn(LearnMsg) returns (Empty) {
    option (gorums.broadcast) = true;
  }
}
```

5.1.1 Broadcast Call

A method with the broadcastcall option will be generated on the Gorums configuration and is intended to be invoked by Gorums clients. These methods are referred to as BroadcastCalls. A BroadcastCall first broadcasts to all the servers in its configuration and collects the responses as they arrive at the client, as shown in Fig. 4.20. Similar to a QuorumCall, it is necessary to define a quorum function to collect the responses. Algorithm 4 shows the signature of a quorum function used for BroadcastCalls.

Algorithm 4 Function Signature of a Quorum Function for a BroadcastCall

```
func (qs QuorumSpec) MethodQF(req *Request, replies []*Resonse) (*Response, bool)
```

5.1.2 Broadcast Option

The broadcast option can be used alone to enable servers with broadcasting functionality or in conjunction with the quorumcall option, creating what we call a QuorumCall with broadcasting enabled. The broadcast framework provides powerful abstractions for requests that need to be persisted across several handlers and servers and need to be related to each other. Therefore, even without all-to-all communication, the broadcast framework can simplify development for the user.

5.2 Broadcast API

The broadcast framework provides new functionality accessible through the broadcast API, which we will highlight in this section. As a baseline when presenting the API, we will assume we have an arbitrary number of clients $\mathcal{C}$ and servers $\mathcal{S}$. A node represents either a client $c \in \mathcal{C}$ or a server $s \in \mathcal{S}$. The broadcast API is primarily targeted towards the server implementation. The client side remains mostly unchanged, with the exception of a few additional manager options, which we will explain in further detail in Section 5.2.2. First, we will introduce the concept of a broadcast request and examine the new server signature. This will be followed by an explanation of how to interact with the broadcast framework and the new functionality it offers.

Broadcast Request

Before delving into the functionality of the broadcast framework, we need to introduce the concept of a *broadcast request*. Rather than treating a client request as a single interaction between a client and a server, we redefine individual requests as messages. This eliminates the traditional request-response pattern and allows nodes to only transmit messages. However, we still adhere to the overall request-response pattern through the notion of broadcast request. A client sends a broadcast request to a server cluster and receives a response from the cluster, as explained in Section 4.3. Consequently, a broadcast request B comprises a set of messages $\mathcal{M}_B$, creating a single request that spans several servers and server handlers. A broadcast request is not limited to the request-response pattern; in some cases, a client or server may only need to broadcast messages without requiring a response. The definition of a broadcast request is thus:

Definition. *A broadcast request B is a request comprising a set of messages $\mathcal{M}_B$ processed by a set of servers $\pi \subseteq \mathcal{S}$, originating from either a client $c \in \mathcal{C}$ or a server $s \in \mathcal{S}$.*

Signature

The first notable change is in the server handler implementation. The signature for both a BroadcastCall and a QuorumCall with broadcasting enabled will be the same. In Go, a regular gRPC server handler is a function that takes in a request and returns a response and an error. However, we modified the signature to include an extra argument and remove the return values, as shown in Algorithm 5.

While we could have retained the return values and allowed a server to reply directly to the last hop (i.e., the calling server or client) as in a regular gRPC invocation, this approach would have turned each broadcast into a QuorumCall, facilitating a one-to-all-to-one pattern for each message. We chose not to do this to maintain the simplicity of the broadcasting functionality and to align it with the group communication specifications explained in Section 2.1.4. Additionally, waiting for replies to a broadcast inside a server handler could easily introduce deadlocks, as using a single lock for multiple handlers would block all subsequent operations until the handler returns.

Algorithm 5 Function Signature of the Broadcast Enabled Server Handlers

```
func (srv *Server) Method(ctx ServerCtx, req *Request, broadcast *Broadcast)
```

5.2.1 Broadcast Struct

As briefly mentioned in the previous section, we have introduced a new argument to the server handler signature named *Broadcast*. This struct, generated by Gorums in the proto files, will be referred to as *broadcast struct* to avoid overloading the term “broadcast”.

The broadcast struct represents the current broadcast request that the incoming message is related to and serves as the main interface for interacting with the broadcast framework. By concentrating most of the functionality into this single broadcast struct, we believe it makes the broadcast framework easy and straightforward to use. A new broadcast struct is generated for each incoming message, with fields populated from the broadcast request and the incoming message, as explained in further detail in Section 5.3.2. These structs are safe for concurrent use and can be stored for later use, which is particularly useful for algorithms with a separate run loop processing messages, such as All-Ack Uniform Reliable

Broadcast [11, p. 126].

Additionally, the broadcast struct contains methods that can be used to perform broadcast actions $\mathcal{A}$ provided by the broadcast framework. A broadcast action $\nu \in \mathcal{A}$ is defined as one of the invocations performed on the broadcast struct. We will introduce these actions in the following sections.

Broadcasting

The broadcast struct enables broadcasting to other servers, which is one of the broadcast actions $\nu_1 \in \mathcal{A}$ provided by the broadcast framework. All methods with the broadcast option defined in the proto specification will become generated methods on the broadcast struct, similar to how methods are generated on the configuration in Gorums. Using the proto specification shown in Listing 5.1, we obtain the following generated methods on the broadcast struct:

- $\mathit{broadcast}.\text{Accept}(\mathit{args})$
- $\mathit{broadcast}.\text{Learn}(\mathit{args})$

Similarly, these methods will be generated on the Gorums configuration, as explained in Section 2.2.4:

- $\mathit{configuration}.\text{Write}(\mathit{args})$
- $\mathit{configuration}.\text{Prepare}(\mathit{args})$

We have removed the arguments and return types for brevity. Notice that we generate the *Write* method on the configuration and not on the broadcast struct. This is because a BroadcastCall is intended to be called only by Gorums clients. Furthermore, BroadcastCall initiates new broadcast requests, whereas the broadcast struct is always related to an existing broadcast request. Invoking a method on the broadcast struct will broadcast the given message to the server's view.

To accommodate initiating broadcast requests from servers, we also generate the methods with the broadcast option on the Gorums server itself, prepending the term "Broadcast":

- $\mathit{server}.\text{BroadcastAccept}(\mathit{args})$

- *server*.BroadcastLearn(*args*)

As presented in Section 4.1.5, the proposer must be able to send accept messages for missing slots. Using the broadcast struct in this case would be incorrect because: 1. The proposer may not have received a broadcast struct corresponding to the broadcast request in the given slot, and 2. A broadcast request is supposed to have a short lifetime and should not be used after it has finished, as explained in Section 5.4.4. Therefore, in these scenarios, the server can use the broadcast methods defined on the server itself. It is important to note that these invocations are broadcast-only and will not accept replies as a regular BroadcastCall would, aligning with the design requirements in Section 4.1.3. As a general rule, servers should use the APIs provided by the broadcast struct. The broadcast methods defined on the Gorums server should only be used when a server needs to create a new, unique broadcast request.

Sending Response

The next broadcast action $\nu_2 \in \mathcal{A}$ provided by the broadcast framework is sending a response to the client. This is a straightforward method that encapsulates the responses for all methods in the proto specification with the BroadcastCall option. Only a single generic method is generated on the broadcast struct:

- *broadcast*.SendToClient(*args*)

It accepts a proto message but does not offer type safety. The type is inferred from the response type of the BroadcastCall method the client invoked. For example, if a client called Write in Listing 5.1, then invoking SendToClient() on a server would send a proto message of type PaxosResponse.

Additional Functionality

We have added extra functionality to the broadcast framework to make it more flexible and to facilitate more protocols. For example, PBFT specifies that a server can forward a client request to the leader [20]. To support this, a few additional methods have been added to the broadcast struct:

- *broadcast*.Forward(*args*)

- $\mathit{broadcast}.\text{To}(\mathit{args})$
- $\mathit{broadcast}.\text{GetMetadata}()$
- $\mathit{broadcast}.\text{Cancel}()$
- $\mathit{broadcast}.\text{Done}()$

The first method, Forward(), takes an address as its only argument. It does exactly what its name suggests: forwards a request to the given address. There are a few constraints: a server can only forward to an address of a server already added to its view, it can only forward messages from clients, and it cannot alter the message. Forwarding is only supported when using a BroadcastCall. These rules are enforced by the broadcast framework, and violations will prevent the message from being forwarded. When these rules are followed, the message will be forwarded to the same gRPC method at the given address, and the broadcast request will be treated as if it was sent directly from the user.

The second method, To(), is meant to be used in conjunction with some of the broadcast methods, such as the accept method in Listing 5.1:

$$\mathit{broadcast}.\text{To}(\mathit{args}).\text{Accept}(\mathit{args}) \tag{5.1}$$

It takes a range of addresses as arguments and lets a user broadcast to specific addresses. These addresses must correspond to servers that have already been added to its view.

The GetMetadata() method returns the metadata related to a broadcast request. These fields are presented in greater detail in Section 5.3.2.

It can be useful to cancel broadcast requests to create latency tail-tolerant systems [61]. Hence, the broadcast framework provides a method named Cancel() that allows the user to broadcast cancellation messages. This is the third broadcast action $\nu_3 \in \mathcal{A}$.

Lastly, the fourth broadcast action $\nu_4 \in \mathcal{A}$, allows the user to signal the end of a broadcast request. This can be used when it is not necessary to return a response to the client. The broadcast framework allocates data structures for each broadcast request, and invoking Done() signals to the broadcast framework that the broadcast request is finished, allowing cleanup to be performed.

5.2.2 Options

The broadcast framework introduces a set of options designed to make it as extensible and flexible as possible. These options are grouped into three categories:

- *Manager Options*: A set of options that enable the user to customize the Gorums client.
- *Server Options*: A set of options that enable the user to customize the Gorums server.
- *Broadcast Options*: A set of options that enable the user to customize the broadcasting functionality of the broadcast framework.

The current version of Gorums already provides ManagerOptions and ServerOptions, but several new options have been added. Only the new options will be briefly presented in this section. When presenting the options, we will include the return type to clearly indicate which category they belong to. We will also remove argument types for brevity. Some options refer to data structures that are presented in greater detail later in this chapter, and thus this list can serve as a reference to the actual implementation of the ideas presented.

Broadcast Server

- **WithShardBuffer(*shardBuffer*)** *ServerOption*: Enables the user to specify the buffer size of each shard. A shard stores a map of broadcast requests. A higher buffer size may increase throughput but at the cost of higher memory consumption. The default is 200 broadcast requests.
- **WithSendBuffer(*sendBuffer*)** *ServerOption*: Enables the user to specify the buffer size of the communication channels to the broadcast processor. A higher buffer size may increase throughput but at the cost of higher memory consumption. The default is 30 messages.
- **WithBroadcastReqTTL(*ttl*)** *ServerOption*: Configures the time-to-live (TTL) a broadcast request should be stored on a server, setting the lifetime of a broadcast processor. The default is 5 minutes.

Broadcasting

- **WithSubset(*srvAddrs*)** *BroadcastOption*: Allows the user to specify a subset of servers to broadcast to. The server addresses given must be a subset of the addresses in the server view.
- **WithoutSelf()** *BroadcastOption*: Prevents the server from broadcasting to itself.
- **AllowDuplication()** *BroadcastOption*: Allows the user to broadcast to the same RPC method more than once for a particular broadcast request.

Identification

- **WithMachineID(*id*)** *ManagerOption*: Enables the user to set a unique ID for the client. This ID will be embedded in broadcast requests sent from the client, making the requests trackable by the whole cluster. A random ID will be generated if not set, which can cause collisions if there are many clients. The ID is bounded between 0 and 4095.
- **WithSrvID(*id*)** *ServerOption*: Enables the user to set a unique ID on the broadcast server. This ID is used to generate BroadcastIDs.
- **WithListenAddr(*addr*)** *ServerOption*: Sets the IP address of the broadcast server, which will be used in messages sent by the server. The network of the address must be a TCP network name.

Connection

- **WithSendRetries(*maxRetries*)** *ManagerOption*: Allows the user to specify how many times the node will try to send a message. The message will be dropped if it fails to send more than the specified number of times. Providing *maxRetries* $= -1$ will retry indefinitely.
- **WithConnRetries(*maxRetries*)** *ManagerOption*: Allows the user to specify how many times the node will try to reconnect to a node. The default is no limit, but it will follow a backoff strategy.

- **WithClientDialTimeout(*timeout*)** *ServerOption*: Enables the user to set a dial timeout for servers when sending replies back to the client in a BroadcastCall. The default is 10 seconds.
- **WithServerGrpcDialOptions(*opts*)** *ServerOption*: Enables the user to set gRPC dial options that the Broadcast Router uses when connecting to a client.

Logging

- **WithLogger(*logger*)** *ManagerOption*: Enables the user to provide a structured logger for the Manager. This will log events regarding the creation of nodes and the transmission of messages.
- **WithSLogger(*logger*)** *ServerOption*: Enables the user to set a structured logger for the Server. This will log internal events regarding broadcast requests.

Authentication

- **WithAllowList(*allowed*)** *ServerOption*: Enables the user to provide a list of (address, publicKey) pairs which will be used to validate messages. Only nodes on the allow list are permitted to send messages to the server, and the server is only allowed to send replies to nodes on the allow list.
- **EnforceAuthentication()** *ServerOption*: Requires that messages are signed and validated; otherwise, the server will drop them.
- **WithAuthentication()** *ManagerOption*: Enables digital signatures for messages.

Execution Ordering

- **WithOrder(*method*$_1$, ..., *method*$_n$)** *ServerOption*: Enables the user to specify the order in which methods should be executed. This option does not order messages but caches messages meant for processing at a later stage. For example, in PBFT, it caches all commit messages if the state is not prepared yet.

- **ProgressTo(*method*$_i$)** *BroadcastOption*: Allows the server to accept messages for the given method or for methods prior in the execution order.

5.3 Broadcast Server

The gRPC framework specifies a client and a server for communication between nodes. In replicated state machine protocols like PBFT [20] and Paxos [30], nodes function both as clients and servers, meaning they can both initiate and handle requests. Thus, a node needs to act as both a client and server in the gRPC framework.

Each client $c \in \mathcal{C}$ and server $s \in \mathcal{S}$ can create a Gorums configuration *config* with any set of servers such that *config* $\subseteq \mathcal{S}$. This *config* object functions as a Gorums client. To support server-to-server communication, we have extended the Gorums server to accept a Gorums configuration *config*. To distinguish it from the Gorums client, we have opted to name the *config* as *view* when used on the server. Similarly, a lightweight server is created on the client side to accept responses from the servers. The client-side server will be explained in more detail in Section 5.5, while this section and the following will shed light on the main logic implemented in the broadcast framework.

The broadcast server manages the entire chain of messages resulting from a broadcast request and is capable of handling client-to-server, server-to-server, and server-to-client communication. It tracks broadcast requests, validates and handles incoming messages, and is responsible for routing responses back to clients.

5.3.1 View

The view is a crucial component of the broadcast server, and the broadcast framework enables the user to set and change views on the server. This can be done by providing a Gorums configuration to the SetView() method on the server generated by Gorums. Broadcasts initiated by a server are constrained to the configured view, giving control to the user and ensuring that messages are not incorrectly routed.

Installing a new view on the server is a significant action because it clears all the current broadcast state to prevent stale data on the server. Additionally, this helps avoid unwanted behavior, such as broadcasting messages related to an old view.

5.3.2 Broadcast Metadata

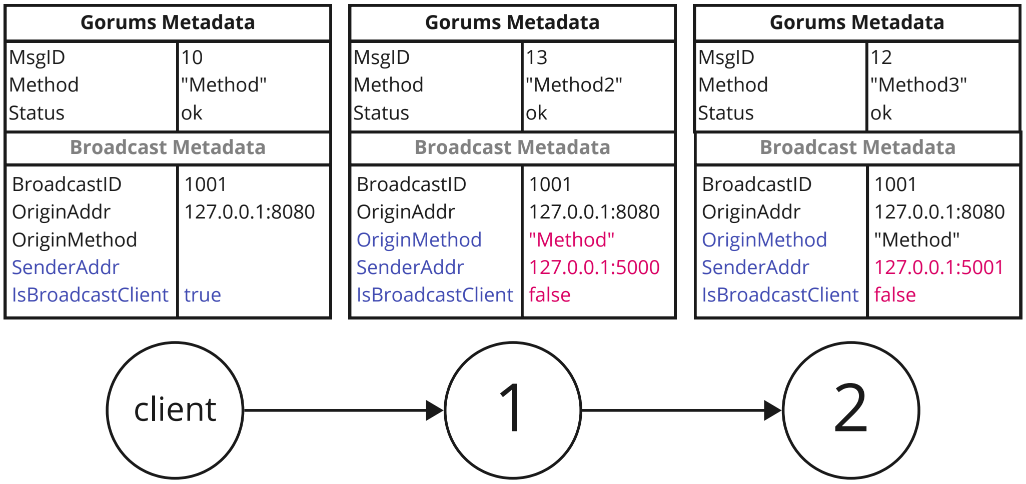
Each message is tagged with metadata, with some fields remaining constant across all servers and handlers while others change based on the sender. The most important field in the metadata is the BroadcastID, which relates different messages from different servers to the same broadcast request. This field is set by the originator $o \in \mathcal{C} \cup \mathcal{S}$ of the broadcast request.

We have enabled two types of broadcast options in the proto specification: broadcastcall and broadcast, as explained in Section 5.1. Two fields in the metadata differentiate these types. The first field is *IsBroadcastClient*, indicating whether the message is sent by a client or a server. The second field is *OriginAddr*, which is the listening address of the client or server that issued the broadcast request. Using these two fields helps differentiate the scenarios where a client either directly connects to all servers, maintaining an established connection for the entire duration of the broadcast request, or sends a message to one server and listens for responses from multiple servers. The former follows the regular request-response pattern in gRPC [62] between the client and all servers and will not contain an address in the *OriginAddr* field. The latter scenario involves a broadcast to a subset of the servers, with the client implementing a client-side server to listen for responses, requiring the *OriginAddr* to be populated in these broadcast requests. The *IsBroadcastClient* field informs the individual servers whether the broadcast request is directly from a client or another server. We will explain how the routing works for both cases in Section 5.4.5.

Lastly, the broadcast metadata contains two additional fields: *SenderAddr* and *OriginMethod*. The *SenderAddr* is the listening address of the last hop or the sender of a message, filled by servers but not clients. The *OriginMethod* field tracks which method the client called and is used only for methods with the broadcastcall option. This field is primarily used to discard malformed messages not related to a client handler.

Fig. 5.1 shows all the fields of the metadata sent with each message. The Gorums metadata is specific to each message (hence inconsistent MsgIDs on each message) and is part of the existing functionality of Gorums, as explained in Section 2.2.2. The broadcast metadata has been added to accommodate all-to-all communication. In the figure, blue fields indicate which fields a particular server can change (this is managed by the broadcast framework and not left to the user), and red fields show examples of what the fields can be.

Note that the metadata sent by a client is more scarce than the metadata between servers; for example, the *OriginMethod* and *SenderAddr* fields are not filled by the client. The *OriginMethod* field is set by a server to correspond to the *Method* field in the Gorums metadata if the message is from a client and the *OriginMethod* field is empty. As shown in the figure, server 2 does not change the *OriginMethod* field because it is non-empty when received from the previous hop, server 1. The *OriginAddr* and *BroadcastID* fields remain the same throughout the broadcast request, while each server sets the *SenderAddr* field to its listening address.



Gorums Metadata	
MsgID	10
Method	"Method"
Status	ok
Broadcast Metadata	
BroadcastID	1001
OriginAddr	127.0.0.1:8080
OriginMethod	
SenderAddr	
IsBroadcastClient	true

Gorums Metadata	
MsgID	13
Method	"Method2"
Status	ok
Broadcast Metadata	
BroadcastID	1001
OriginAddr	127.0.0.1:8080
OriginMethod	"Method"
SenderAddr	127.0.0.1:5000
IsBroadcastClient	false

Gorums Metadata	
MsgID	12
Method	"Method3"
Status	ok
Broadcast Metadata	
BroadcastID	1001
OriginAddr	127.0.0.1:8080
OriginMethod	"Method"
SenderAddr	127.0.0.1:5001
IsBroadcastClient	false

Figure 5.1: The figure illustrates the fields in the metadata sent in a broadcast request. The Gorums metadata changes for each message and is part of the existing Gorums functionality. The broadcast metadata has been added to support all-to-all communication. In the figure, the blue fields indicate which fields the servers can change, while the red fields show examples of potential values.

5.4 Broadcast Manager

The broadcast manager contains the main logic of the broadcasting functionality. It serves as the core of the broadcast server, maintaining the state and routing of broadcast requests. The following sections will delve into the responsibilities of the manager, starting with how asynchronous messaging is enabled and handled. Next, we will introduce the concept of BroadcastID and then explain how sharding is implemented. Following that, we will provide a detailed overview of the broadcast processor and, finally, present the broadcast router.

5.4.1 Asynchronous Messaging

All messages received from a single client c are run synchronously by default in Gorums to ensure the correct ordering of messages. This approach also has the benefit of potentially not requiring a mutex in the implementation. However, it can significantly slow down broadcast requests, as these can theoretically be processed in parallel. To address this, a broadcast handler has been implemented that wraps the server handler and runs it asynchronously. Each message is processed in a separate goroutine, and the Gorums server mutex is released immediately to allow the server to process the next message.

Several considerations are taken into account when designing and enabling asynchronous messaging. By not being careful it is easy to create deadlocks or goroutine leaks. A simple solution to prevent race conditions is to use a single mutex in each server handler, ensuring synchrony between handlers. However, if implemented incorrectly, a deadlock could occur if a server from one handler is broadcasting to another handler on itself. This could cause the first handler to block until it has processed the broadcast, while subsequent messages would have to wait for the handler to return before being processed. Consequently, all handlers and broadcasts run asynchronously by default. The drawback of this approach is that the message ordering is not maintained as they are in the current version of Gorums.

5.4.2 BroadcastID

The BroadcastIDs have been carefully designed, inspired by SnowflakeIDs created by X (formerly known as Twitter) [63], to maximize concurrency and parallelism. Initially, we experimented with using UUIDs for BroadcastIDs because each BroadcastID must be unique. UUIDs, based on RFC 4122 [64], are sortable by time and can be used as keys in maps for direct comparison. However, at 128 bits, these IDs are unnecessarily long for our use case. Therefore, we opted to

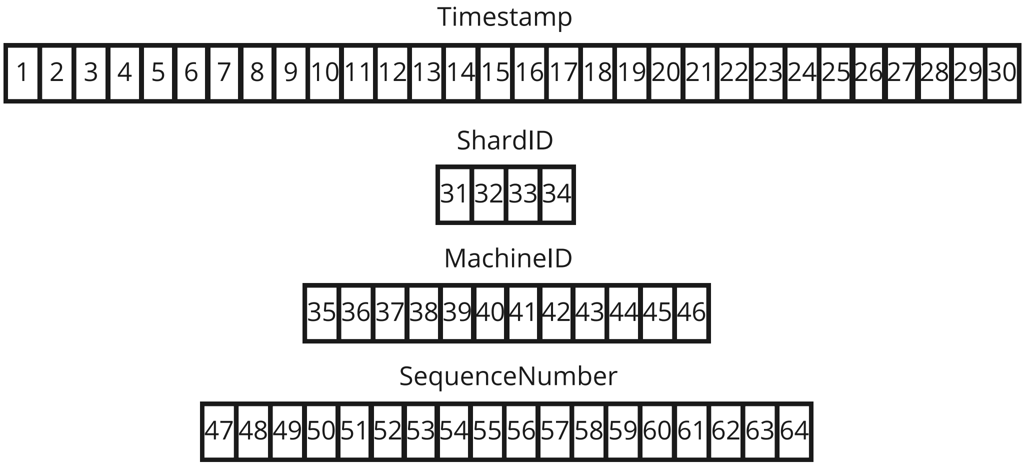


Figure 5.2: Structure of the BroadcastID

create a custom version of SnowflakeIDs, providing a 64-bit unique identifier for each broadcast request.

A broadcast request is intended to be short-lived, as further explained in Section 5.4.4, making a 64-bit identifier sufficient. Fig. 5.2 illustrates the structure of the ID: 30 bits for the timestamp in seconds since January 1st, 2024, 4 bits for the ShardID, 12 bits for the MachineID, and 18 bits for the SequenceNumber. The MachineID is unique to each client and can be provided by using the WithMachineID() manager option. Otherwise, a random MachineID will be generated. The ShardID specifies which shard should handle the request, as explained in Section 5.4.3. The SequenceNumber increments with each broadcast request sent by the client, indicating the number of messages sent. It rolls over upon reaching its maximum value and restarts at zero, meaning it facilitates a large number of broadcast requests per second. Currently, a single client can send up to about a quarter million unique broadcast requests per second per shard.

5.4.3 Sharding

Module 6 Interface and Properties of the Broadcast Shard

Module:

Name: Broadcast Shard, **instance** *bs*.

Events:

Request: $\langle$ *bs*, *Process* | $id,\ p,\ message$ $\rangle$: Requests the shard to send the incoming $message$ from client or server p to broadcast processor P with $P.id = id$.

Indication: $\langle$ *bs*, *Deliver* | $response$ $\rangle$: Indicates that the shard has handled a message with output $response$.

Request: $\langle$ *bs*, *Send* | $id,\ \nu,\ \pi,\ message$ $\rangle$: Requests the shard to send the incoming action ν and the $message$ to broadcast processor P with $P.id = id$.

Properties:

BS1: *Independence:* A message m_1 related to broadcast request B_1 and message m_2 related to broadcast request B_2 will be processed asynchronously and separately on a server s.

Inspired by database technology [65], we have implemented shards to distribute the work performed by the servers and enable horizontal scaling to some extent. Shards are responsible for caching, managing, and tracking broadcast processors throughout their lifetime. The ShardID is included in the BroadcastID, as described in the previous section, allowing multiple shards to run on each server. Each shard uses an RWMutex for concurrency, as there are usually many more reads compared to writes. Furthermore, each shard operates independently and can run in parallel, thereby improving performance. The number of shards is configurable and can vary for each server, within the range of 1 to 16.

Incoming messages to a server srv are routed by the broadcast manager to the appropriate shard. The Process event is then called on the shard, which either creates a broadcast processor P if one does not exist or passes the message along to P if it does. Broadcast processors will be explained in detail in the next section; for now, it suffices to say they are the compute units that handle incoming messages. Communication between the shard and P occurs through two Go channels: one for forwarding incoming messages from clients $\mathcal{C}$ or servers $\mathcal{S}$ to P and the other for forwarding actions invoked by the user-defined logic on srv.

5.4.4 Broadcast Processor

The broadcast processor is the most fundamental component of the broadcast framework, encompassing most of its logic. It is responsible for fulfilling most of the requirements outlined in the broadcast specification shown in Module 3. The broadcast processor allows the user to perform broadcasts, send cancellations, and respond to the client. Additionally, it tracks which methods a server has broadcast to and manages the execution order of incoming messages.

Module 7 Interface and Properties of the Broadcast Processor

Module:

Name: Broadcast Processor, **instance** *bp*.

Events:

Request: ⟨ *bp*, *Process* | $p,\ message$ ⟩: Requests the broadcast processor to handle the incoming $message$ from client or server p.

Indication: ⟨ *bp*, *Deliver* | $response$ ⟩: Indicates that the broadcast processor has handled a message with output $response$.

Request: ⟨ *bp*, *Broadcast* | $h,\ \pi,\ message$ ⟩: Requests the broadcast processor to execute a broadcast to handler h on the set of servers π with the given $message$.

Request: ⟨ *bp*, *Cancel* | $\pi,\ cancellation$ ⟩: Requests the broadcast processor to broadcast a $cancellation$ to the set of servers π.

Request: ⟨ *bp*, *SendToClient* | $c,\ response$ ⟩: Requests the broadcast processor to execute a unicast to the client c with the given $response$.

Request: ⟨ *bp*, *Done* | $cancellation$ ⟩: Requests the broadcast processor to stop and finish the execution.

Properties:

BP1: *Termination:* A message m processed by broadcast processor P will eventually return a $response$.

BP2: *Execution ordering:* A message m corresponding to handler H_n will be cached until the order $\psi \geq n$ if execution order is enforced.

BP3: *Uniqueness:* If a broadcast processor P_1 is created for broadcast request B_1 and a broadcast processor P_2 is created for broadcast request B_2, then $B_1 \neq B_2$.

BP4: *Relation:* If a broadcast request B is processed by broadcast processor P, then $P.id = B.id$.

BP5: *Consistency:* If a message m related to broadcast request B is processed by broadcast processor P_1 and broadcast processor P_2, then $P_1 = P_2$.

BP6: *No duplication:* No message is broadcast more than once.

BP7: *Validity:* No message m related to a broadcast request B with broadcast processor P such that $B.finished = true$ will be processed by P.

Whenever a new broadcast request B arrives at a server srv, a broadcast processor P is created with $P.id = B.id$ and lifetime ϵ such that $P.\epsilon = B.\epsilon$. P is initialized with state θ, which includes information such as OriginAddr and SenderAddr related to B. Furthermore, θ tracks the methods P has broadcast to and whether P has sent a $cancellation$ for B. After P is created, the message m related to B will be processed by P using the ***Process*** event. As B comprise a set of messages $\mathcal{M}_B$, an incoming message $m \in \mathcal{M}_B$ where srv has not yet seen B does not need to be sent directly from the origin $o \in \mathcal{C} \cup \mathcal{S}$ nor be the first message in the broadcast request. This facilitates the broadcast request spanning several handlers and servers, allowing individual servers to participate in algorithms without receiving the original message from the client. Furthermore, P is responsible for processing all incoming messages $\mathcal{M}_B$ and returning an indication *Deliver*. P processes m before the user implemented server handler h is run, enabling P to drop m if it is not valid or if B is finished, therefore preventing h to be wrongfully executed. This is done by returning an error as $response$ in the indication. Otherwise, P will return a cancellation context for B and a function that can be used to perform broadcast actions. A broadcast action ν is an invocation on the broadcast struct as described in Section 5.2.1 and can be categorized as either a *Broadcast*, *Cancel*, *Done*, or a *SendToClient* event.

A more detailed description of what each event does is outlined in Algorithm 8. The broadcast event enables sending m to server $p \in \pi \subseteq \mathcal{S}$, hence enabling the all-to-all communication pattern by letting each server send a one-to-all broadcast. It also provides additional flexibility by enabling a server to broadcast to only a subset of the servers in the view. Each server can decide to broadcast a unique message m for all broadcast enabled handlers $\mathcal{H}_B = \{h \in \mathcal{H} : h.broadcast = true\}$ where $\mathcal{H}$ are all methods defined in the proto specification as described in Section 5.1.

The Cancel event broadcasts a special cancellation message to servers π, signifying that the broadcast request may be finished. However, it is up to the user implementation to decide how to handle cancellations. When a cancellation is received, the ServerCtx provided in the server handler will be canceled. This allows the user to ignore the context if desired, meaning cancellations will not affect the system beyond the overhead of transmitting a few extra messages.

The SendToClient event first establishes a connection to the client and then sends the response. This marks the termination of the broadcast request, and consequently, the corresponding broadcast processor is stopped. Each broadcast processor has an assigned context, which is canceled when the broadcast processor stops. It is important to note that this context is different from the cancellation context. This allows the shard to use the context to determine whether it should still accept new messages related to the broadcast request.

Running a separate broadcast processor for each broadcast request enables simultaneous processing of multiple messages and actions, reducing unnecessary waiting and promoting parallelism. When ordering is not needed, only the first message of a new broadcast request and the client's message need to be processed, significantly cutting down on processing time. Meaning, that for most incoming messages, the shard only needs to verify the existence and if the broadcast processor is still running. The broadcast processor abstracts many intricate implementation details, contributing to making the broadcast framework user-friendly and transparent.

Algorithm 8 Broadcast Processor Implementation

```
**Implements:**
  BroadcastProcessor, **instance** *bp*.
**Uses:**
  Router, **instance** *r*;
**on** $\langle\, rsm, \mathit{Init} \,\rangle$
  $\mathit{state} \leftarrow$ Initial state
  $\mathit{state.sentCancellation} \leftarrow$ **false**
  $\mathit{state.cache} \leftarrow \varnothing$
**on** $\langle\, bp, \mathit{Process} \mid p, m \,\rangle$
  **if** $\neg \mathit{isValid}(m)$ **then**
    **return**
  **if** $\mathit{typeOf}(m) = \mathit{cancellation}$ **then**
    $\mathit{cancelServerCtx}()$        ▷ Cancels the ServerCtx that is used in the server handlers
    **return**
  **if** $\neg \mathit{isInOrder}(m)$ **then**
    $\mathit{state.cache} \leftarrow \mathit{state.cache} \cup m$        ▷ Caches the message until later
    **return**
  $(\mathit{newstate}, \mathit{response}) \leftarrow \mathit{update}(m, \mathit{state})$
  $\mathit{state} \leftarrow \mathit{newstate}$
  **trigger** $\langle\, bp, \mathit{Deliver} \mid \mathit{response} \,\rangle$
**on** $\langle\, bp, \mathit{Broadcast} \mid h, \pi, m \,\rangle$
  **if** *alreadyBroadcasted(h)* **then**
    **return**
  **trigger** $\langle\, r, \mathit{Broadcast} \mid \pi, m, \mathit{bp.id} \,\rangle$
  **if** $\mathit{state.cache} \neq \varnothing$ **then**
    $\mathit{handleOutOfOrder}(\mathit{state.cache}, h)$        ▷ Dispatch out of order msgs corresponding to *h*
**on** $\langle\, bp, \mathit{Cancel} \mid \pi \,\rangle$
  **if** *state.sentCancellation* **then**
    **return**
  $\mathit{state.sentCancellation} \leftarrow$ **true**
  **trigger** $\langle\, r, \mathit{Cancel} \mid \pi, \mathit{bp.id} \,\rangle$
**on** $\langle\, bp, \mathit{SendToClient} \mid c, resp \,\rangle$
  **trigger** $\langle\, r, \mathit{Send} \mid c, resp, \mathit{bp.id} \,\rangle$
  $\mathit{state} \leftarrow \varnothing$
  $\mathit{exit}()$        ▷ Stops the processor and marks it ready to be garbage collected
**on** $\langle\, bp, \mathit{Done} \,\rangle$
  $\mathit{state} \leftarrow \varnothing$
  $\mathit{exit}()$        ▷ Stops the processor and marks it ready to be garbage collected
```

Execution Ordering

The broadcast framework can be configured to guarantee the execution ordering of messages, meaning it is possible to specify which handler should be run and in what order. This can be helpful in algorithms that require the acquisition of a specific message before executing the next step in the algorithm such as in PBFT.

The execution order can be specified when creating the server using the WithOrder() option. In PBFT the execution order would be as follows: PrePrepare, Prepare, Commit. The framework will then guarantee that a server will not run the prepare handler before the pre-prepare handler has run, and similarly, the commit handler will not run before the prepare handler. A server will only proceed to the next stage by either using the broadcast option ProgressTo() or by invoking a method that is later in the execution order. More generally, we can define a system where there is a specific sequence of handlers that need to be executed. We can denote this sequence of handlers $\bar{H}$, which is defined as:

$$\bar{H} = \{h_0, h_1, \ldots, h_n\} \subseteq \mathcal{H}_B \tag{5.2}$$

where $\mathcal{H}_B$ is the set of all broadcast enabled handlers. We use an order index ψ, initialized at $\psi = 0$, to determine which handler in the sequence is allowed to be executed. Also, $\bar{H}$ is arranged in ascending order, meaning h_i will be executed before h_j if $i < j$. Following the notation from the previous section, $\mathcal{M}_B$ is the set of messages related to broadcast request B. Any message $m' \in \mathcal{M}_B$ with execution order $m'.\psi > \psi$ will be temporarily stored until the server invokes a handler $h_x : x \geq m'.\psi$. Messages to handlers $H' \not\subset \bar{H}$ will always be executed immediately.

Cancellations

Cancellations can improve performance [61] by reducing the amount of work needed. Due to the nature of unreliable servers and communication, only a weak guarantee is provided for cancellations. This means that a client can cancel a request by broadcasting a cancellation, but there are no consensus mechanisms or additional measures to ensure delivery of the cancellation to all nodes. For instance, a client might only know a single server, such as the leader, and send a broadcast request only to this server. If the leader fails mid-execution, the other servers have no way of knowing if the client cancels the request because the client communicates solely with the leader. Guaranteeing a cancellation would require all nodes to agree on whether to cancel the request, which is not feasible in this particular scenario. It is thus left to the user to enforce and handle cancellations, meaning cancellations are not enabled by default. To enable client-side cancellations, the user must provide the *cancelOnTimeout* option when invoking a BroadcastCall. On the server side, the user must listen to the ServerCtx and use *broadcast*.Cancel() to propagate cancellations.

Request lifetime

A broadcast request is considered finished based on the following criteria:

- *SendToClient event*: The user can invoke the SendToClient event when the broadcast request is complete, sending a response back to the client.
- *Done event*: The user can invoke the Done event when a broadcast request is finished, but sending a response to the client is not necessary.
- *The broadcast processor's TTL has expired*: The user can specify a TTL for a broadcast request and its broadcast processor. This ensures that the broadcast processor will stop within the specified time limit. The default TTL is set to 5 minutes.

Garbage Collection

Since the lifetime of a broadcast request spans several handlers and servers, it is necessary to cache data temporarily. This means that state is stored on each server, which could lead to stale data if implemented poorly. To manage this, most of the temporary data is stored and managed by a struct named state. The state is responsible for creating shards and cleaning up when the server is stopped or if a view change occurs, as described in Section 5.3.1.

To keep track of incoming and handled requests, the broadcast processors are persisted even after they are stopped. To prevent out-of-memory errors, a garbage collection mechanism has been implemented. A broadcast processor can only be removed after the TTL, regardless of whether it finished early. This ensures that subsequent messages with a previously removed BroadcastID are not added to servers again. To reduce the memory footprint of the broadcast processor after it has stopped, most of the data structures are removed and cleaned up by the Go garbage collector when the broadcast processor stops. After the TTL, the broadcast processor is removed from the shard map and fully cleaned up.

5.4.5 Router

Algorithm 9 Broadcast Router Implementation

```
Implements:
    Router, instance $r$.
Uses:
    GorumsChannel, instance $ch$;
on $\langle\, r, \mathit{Init} \,\rangle$
    $\mathit{clientConns} \leftarrow \varnothing$
on $\langle\, r, \mathit{Broadcast} \mid \pi, m, id \,\rangle$
    for all $p \in \pi$
        trigger $\langle\, \mathit{ch}, \mathit{Send} \mid p, m, id \,\rangle$
on $\langle\, bp, \mathit{Cancel} \mid \pi, id \,\rangle$
    for all $p \in \pi$
        trigger $\langle\, \mathit{ch}, \mathit{Send} \mid p, cancel, id \,\rangle$
on $\langle\, bp, \mathit{SendToClient} \mid c, resp, id \,\rangle$
    $\mathit{clientConn} \leftarrow \mathit{clientConns}[c]$        ▷ A clientConn is a GorumsChannel instance
    if $\neg\mathit{clientConn}$ then
        $\mathit{clientConn} \leftarrow \mathit{createConn}(c)$
        $\mathit{clientConns} \leftarrow \mathit{clientConns} \cup \mathit{clientConn}$
    trigger $\langle\, \mathit{clientConn}, \mathit{Send} \mid c, resp, id \,\rangle$
```

The router is responsible for sending messages to nodes and the correct handlers, routing outgoing broadcast messages, cancellations, and client responses. A client can either connect directly to a server or tag the metadata with its address, as explained in Section 5.3.2. The fields *IsBroadcastClient* and *OriginAddr* in the broadcast metadata determine how to route the message. If *OriginAddr* is empty, the client must have connected directly to the server for a response, indicating that QuorumCall with the broadcast option is used. In this case, servers that the client does not connect to directly (e.g. if the client only sends a message to the leader) will drop the response message while still considering the broadcast request successful. If *OriginAddr* is non-empty, each server will create a connection to the client-side server and send the response back to the client. Connections to clients are persisted to increase performance. Algorithm 9 shows the logic behind the router.

Retries

There are several reasons why a message in transit might be dropped or not delivered, such as network partitions, congestion, crashes, or timeouts [12][20]. To achieve reliable messaging, protocols must implement retry mechanisms or account for these issues at the algorithm level, for example, by allowing failures. TCP [66], used by Gorums, guarantees delivery, so lost messages are typically due to server crashes or timeouts caused by network issues. By default, a node in Gorums will only attempt to send a message once to another node and will drop it on failure. Nodes maintain connections to each other while online and have reconnection capabilities if a node goes offline, ensuring message delivery when nodes are available. Messages are not stored for later delivery to prevent network flooding and because of their short lifespan, reducing the need for cleanup compared to storing them. This reliability implementation is already provided by the current version of Gorums and is utilized in the broadcast framework. This is reflected in Algorithm 9, where a Gorums channel is used to send messages. However, some protocols may require that messages are eventually delivered or have bounded delivery times. For this, a manager option WithRetries() has been implemented, allowing the specification of the number of retry attempts for a message. If the limit is exceeded, the message is dropped. Setting WithRetries(-1) will cause the application to attempt to send messages indefinitely. It is configured with the same backoff strategy as the reconnection functionality described in Section 2.2.2.

5.5 Client Server

The client-side server enables a client to collect responses for a broadcast request without needing to know the addresses of all servers in a view. The client-server registers a handler for each method with the broadcastcall option in the proto specification. Incoming messages specify which handler should process them by using the *Method* field in the metadata sent with the message, similar to how the Gorums server operates. The client-server requires the user to provide a listener and an address. This address is used in the *OriginAddr* field in the metadata sent in broadcast requests by the client.

The client-server primarily contains two methods:

- AddRequest()
- AddResponse()

The AddRequest() method creates a broadcast request and prepares the client-server to handle responses from the server cluster. To facilitate running several broadcast requests asynchronously, the method starts a goroutine for each broadcast request. Responses from the servers are sent to and processed by the corresponding goroutine. The AddResponse() method is responsible for routing the responses to the appropriate goroutine. It is invoked each time a handler on the client-server is run, and it ensures that outdated responses and those not belonging to a broadcast request are dropped.

5.6 Security Analysis

One of our goals is to make the broadcast framework as secure as possible, so it has been designed with security in mind. In this section, we will perform a security analysis of the broadcast framework, identify potential attack surfaces, and explain how the broadcast framework can be configured to mitigate these attacks.

In some server configurations, it is possible for a client to only know a subset of the servers, but BroadcastCall allows all servers in the cluster to respond to the client. If some of the intermediate servers are Byzantine, they can forge requests from the client and broadcast them to the other servers. To counter this, digital signatures are an option in the configuration, where the original message is signed with the client's private key. The client's public key and a hash of the original message are then added to the metadata of the broadcast message, allowing all downstream servers to validate the original message.

A client can cause a Server-Side Request Forgery attack by sending an arbitrary origin address. To mitigate this, a whitelist of IP addresses can be provided to the servers by using the WithAllowList() server option, ensuring that servers only reply to these allowed addresses.

The *OriginMethod* field in the broadcast metadata indicates which handler the client sent a message to, as explained in Section 5.3.2. Modifying this field could prevent servers from sending a response back to the client, enabling a denial of service attack. To prevent this, the entire message, including all metadata, is included in the signature. This way, all servers can validate the *OriginMethod* field when checking the signature of a message.

Enabling cancellations introduces another attack surface. A server can issue a cancellation, and there is no way for the other servers to fully trust that the broadcast request was indeed canceled by a client. This is because the client can choose to send the initial broadcast request to only one server, such as the leader, and not to all servers. There is no simple solution to prevent this type of attack, so cancellations only notify that a broadcast request can be canceled. It is up to the user to decide whether to adhere to the cancellation.

Each broadcast request is required to have a unique identifier called BroadcastID, as explained in Section 5.4.2. A Byzantine node can send messages with BroadcastIDs belonging to other broadcast requests. Consequently, servers can end up with different metadata for a single broadcast request, causing replies to be routed incorrectly. To mitigate this issue, the metadata could be based on a quorum of messages or a prioritization list. This would require temporarily storing all messages related to a broadcast request on each server. Currently, the broadcast framework does not implement this functionality, which is a shortcoming. This task is left for future work, where server handlers can be designed to act as quorum functions, allowing for a reevaluation and potential patching of this attack vector.

A server can receive messages from any source. It is not possible for a server to differentiate between a server and a client other than by checking the *IsBroadcastClient* field in the metadata, which can be altered in a forged message. Also, a server will accept messages regardless of whether the sending server is participating in a specific protocol. Therefore, it is essential to define a set of trusted nodes on a server and only trust messages from this set. The broadcast framework allows the user to provide a whitelist to the servers and the client-servers, which will only allow messages from the specified nodes.

Another attack is man-in-the-middle attacks. Since servers can act as intermediaries, they can alter messages in transit or pretend to be the owner of another server's address. To mitigate this, the addresses of the servers are included in the signature when signing messages. gRPC also supports TLS encryption [67], which helps prevent man-in-the-middle attacks aimed at intercepting traffic.

There are two attack surfaces in the broadcast framework that have not been addressed: rate limiting and nodes refusing to forward or send messages, which can lead to denial-of-service and prevent progress in algorithms, respectively. Rate limiting is thus identified as an area for further work. While the broadcast framework cannot directly solve the problem of reluctant nodes, developing primitives to address such scenarios could be considered.

Chapter 6

Practical Application

In this chapter, we will outline how the broadcast framework can be used to implement algorithms with the all-to-all communication pattern through a simple example.

6.1 Eager Reliable Broadcast

In this section, we will demonstrate how to implement a server and a client that use the broadcast functionality provided by the broadcast framework. We will implement the Eager Reliable Broadcast algorithm given in [11, p. 124]. Fig. 6.1 provides an overview of our goal: a client broadcasts a request to the servers, which in turn broadcast *Deliver* to all servers in their view, and finally, all servers respond to the client. The proto specification used in the example implementation is shown in Listing 6.1.

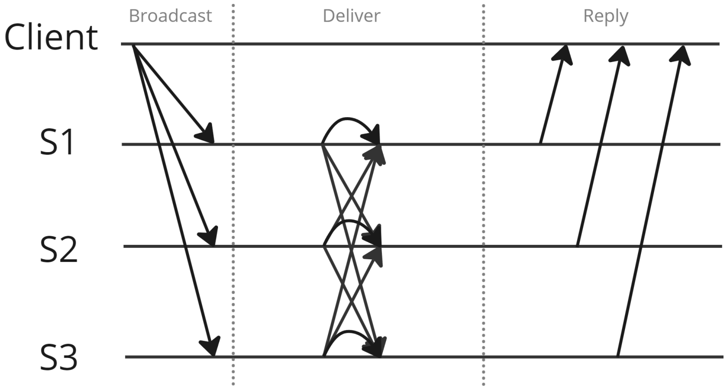


Figure 6.1: Overview of the Eager Reliable Broadcast [11, p. 124] implementation.

Listing 6.1 : The proto specification used in the example implementation.

```
service ReliableBroadcast {
  rpc Broadcast(Message) returns (Message) {
    option (gorums.broadcastcall) = true;
   }
  rpc Deliver(Message) returns (Empty) {
    option (gorums.broadcast) = true;
   }
}

message Message {
  string Data = 1;
}

message Empty {}
```

6.1.1 Client

The creation of a client using the broadcast framework is almost identical to how it is done in the current version of Gorums. Listing 6.2 shows a complete code snippet on how to create the client, with errors ignored for brevity. To create the client, we need to create a Gorums Manager, a Gorums configuration, and a QuorumSpec. The implementation of the QuorumSpec is shown in Listing 6.3. This process is the same as in the current version of Gorums. However, the broadcast framework introduces a new method on the Gorums Manager: AddClientServer(). This method creates and starts the client-side server needed to collect responses. It takes two arguments: a listener and an address. The implementer must create the listener, providing flexibility and not hiding important implementation details such as which addresses and ports a client is listening to. The address is used in the metadata, as explained in Section 5.5, and the listener is used by the underlying gRPC server.

Listing 6.2 : How to create a client.

```go
type Client struct {
    mgr    *pb.Manager
    config *pb.Configuration
}

func New(addr string, srvAddresses []string, qSize int) *Client {
    mgr := pb.NewManager(
        gorums.WithGrpcDialOptions(
            grpc.WithTransportCredentials(insecure.NewCredentials()),
        ),
    )
    lis, _ := net.Listen("tcp", addr)
    mgr.AddClientServer(lis, lis.Addr())
    config, _ := mgr.NewConfiguration(
        NewQSpec(qSize),
        gorums.WithNodeList(srvAddresses),
    )
    return &Client{
        mgr:    mgr,
        config: config,
    }
}

func (c *Client) Broadcast(ctx context.Context, value string) (*pb.Message,
    error) {
    req := &pb.Message{
        Data: value,
    }
    return c.config.Broadcast(ctx, req)
}
```

Listing 6.3 : How to create a QuorumSpec.

```go
type QSpec struct {
    quorumSize int
}

func NewQSpec(qSize int) pb.QuorumSpec {
    return &QSpec{
        quorumSize: qSize,
    }
}

func (qs *QSpec) BroadcastQF(in *pb.Message, replies []*pb.Message) (*pb.Message
    , bool) {
    if len(replies) < qs.quorumSize {
        return nil, false
    }
    return replies[0], true
}
```

With the prerequisites implemented, it is possible to send a broadcast request. This request will be sent to all the servers in the Gorums configuration, and the replies will be processed by the quorum function *BroadcastQF*, which returns a single reply from the BroadcastCall.

6.1.2 Server

Creating a server is relatively simple and closely follows the process in the current version of Gorums. Listing 6.4 shows all the steps involved in creating a server. First, we need to embed the generated Gorums server implementation. Although it is possible to define it as a property, embedding the server makes it easier to use and more readable. To use the functionality provided by the broadcast framework, we need to provide the server option WithListenAddr() when creating the server. This option populates the *SenderAddr* field in the metadata, as described in Section 5.3.2. Next, we need to assign a view to the server using the SetView() method, which accepts a Gorums configuration. The addresses of the other servers are the only required parameters to facilitate communication between the servers.

Listing 6.4 : How to create a server.

```go
type Server struct {
    *pb.Server
    mut       sync.Mutex
    delivered []*pb.Message
    mgr       *pb.Manager
    srvAddrs  []string
}

func New(lis net.Listener, srvAddrs []string) *Server {
    srv := Server{
        Server:   pb.NewServer(gorums.WithListenAddr(lis.Addr())),
        srvAddrs: srvAddrs,
        delivered: make([]*pb.Message, 0),
    }
    srv.configureView()
    pb.RegisterReliableBroadcastServer(srv.Server, &srv)
    go srv.Serve(lis)
    return &srv
}

func (srv *Server) configureView() {
    srv.mgr = pb.NewManager(
        gorums.WithGrpcDialOptions(
            grpc.WithTransportCredentials(insecure.NewCredentials()),
        ),
    )
    view, _ := srv.mgr.NewConfiguration(gorums.WithNodeList(srv.srvAddrs))
    srv.SetView(view)
}
```

Listing 6.5 shows how to define the handlers on the server and the actual implementation of the Eager Reliable Broadcast algorithm. Following the proto specification shown in Listing 6.1, we must implement the methods *Broadcast* and *Deliver*. The server handler *Broadcast* will run when the client invokes the BroadcastCall. The server will then proceed to broadcast *Deliver* to the other servers. In the *Deliver* handler, the server will first check if it has already delivered the message. If not, it will broadcast *Deliver* again and then reply to the client. It is important to note that the broadcast framework prohibits the servers from broadcasting *Deliver* more than once for each broadcast request.

Listing 6.5 : How the server handlers are defined.

```
func (s *Server) Broadcast(ctx gorums.ServerCtx, request *pb.Message, broadcast
    *pb.Broadcast) {
    broadcast.Deliver(request)
}

func (s *Server) Deliver(ctx gorums.ServerCtx, request *pb.Message, broadcast *
    pb.Broadcast) {
    s.mut.Lock()
    defer s.mut.Unlock()
    if !s.isDelivered(request) {
        s.delivered = append(s.delivered, request)
        broadcast.Deliver(request)
        broadcast.SendToClient(request, nil)
    }
}

```

```
func (s *Server) isDelivered(message *pb.Message) bool {
    for _, msg := range s.delivered {
        if msg.Data == message.Data {
            return true
        }
    }
    return false
}
```

6.1.3 Best Practices

The broadcast functions should be placed at the end of a server handler. This is because the broadcast invocation will block until it has queued a message for transmission, potentially allowing the next handler to be processed before returning from the current handler. Furthermore, it is bad practice to broadcast more than once in the same handler. While it is acceptable to define several paths and outcomes, broadcasting multiple times from one handler can lead to unwanted side effects, as using broadcast.To() can slightly mutate the broadcast struct. However, broadcasting to a method and sending a reply to the client can be done within the same method. Since SendToClient() stops execution, it is recommended that this is always run last.

Chapter 7

Experimental Evaluation

Providing tools that simplify the implementation of distributed algorithms and reduce errors can lead to overhead and performance trade-offs. In this chapter, we benchmark two algorithms, Paxos and PBFT, comparing their performance to baseline implementations. First, we discuss the experimental setup and the dataset used. Next, we present the benchmark results and analyze their implications. We also highlight a design choice made to optimize the performance of the broadcast framework.

7.1 Experimental Setup and Data Set

All tests are conducted on virtual machines provided by the Azure cloud computing platform [68]. These virtual machines are collocated within the same data center (Norway East), but by utilizing availability zones, we ensure they run on separate physical hardware [69]. Throughout all experiments, we use a single virtual machine running 10 Gorums clients and between 3-4 servers each running on a separate virtual machine. Table 7.1 details the types of virtual machines we use for clients and servers. The Dsv3 family, known for their balanced CPU-to-memory ratio, is ideal for both testing and production workloads [70].

During all benchmarks, we monitor the resource utilization of the client machine, ensuring that memory and CPU usage remain well below the 32 GiB maximum and CPU limit, respectively. This guarantees that the client machine is not a bottleneck in the benchmarks. To minimize IO operations, we only store data in-memory, avoiding disk writes on the servers. Additionally, the Go garbage collector is enabled throughout all benchmarks.

Type	Instance	vCPU	Memory (GiB)
Client	Standard D8s v3	8	32
Server	Standard D4s v3	4	16

Table 7.1: The client machine is configured with a standard D8s v3 while the servers are configured with standard D4s v4 virtual machines offered by Azure[70].

To measure the latency versus the throughput of the systems, we configure the clients to send a target throughput of requests per second to the servers. Each request corresponds to a full write operation, and the latency is measured as the round-trip time of a request from the client and back. All benchmarks are run five times, and the average of all runs is used to produce the results.

Additionally, we run benchmarks to measure performance under normal operation. In these tests, each client is configured to send 100 requests, and the processing times for each request are measured. These tests are run 10 times, and the average processing times from these runs are used to generate the results.

The algorithms Paxos and PBFT are used to evaluate the broadcast framework. All tests are implemented without leader election to minimize overhead. A leader is selected at the start of the tests and remains unchanged throughout the benchmarks. Therefore, in Paxos, only the accept, learn, and commit phases are executed. In PBFT, all three phases—pre-prepare, prepare, and commit—are executed, but the view change and garbage collection logic are disabled. Additionally, authentication and TLS are not implemented in any of the tests, as the extra overhead from these protocols would be consistent across implementations and thus does not affect the relative performance comparison between the baselines and the implementations using the broadcast framework.

The data-centric version of Paxos has been implemented using QuorumCall and serves as a baseline.[1] It has been carefully designed to minimize locking and maximize request processing speed by handling them in parallel if possible. Although this version does not utilize the all-to-all pattern, the broadcast framework can still simplify the development of the Paxos protocol. In this case,

[1] This implementation is taken from a private Github repository (`https://github.com/dat520/dat520/tree/main/lab5/gorumspaxos`) and is developed by Hein Meling and Hanish Gogada.

the broadcast framework assists in managing the lifetime of requests and alleviates issues related to delayed message deliveries and deadlocks. Additionally, a process-centric version has been implemented using the broadcast framework and is compared to the two data-centric versions.

Implementing PBFT without modifications in the current versions of Gorums has not been possible, so a version using only Go and gRPC is implemented and tested against a version using the broadcast framework. The first is called the *plain* version and the latter version is simply called *Gorums*. The goal for the plain version is to create a straightforward, naive implementation without complex optimizations, thus prioritizing simplicity. However, to ensure a more fair comparison, some optimizations were included such as reusing connections between nodes, locking resources only when necessary, and running network calls in separate goroutines.

Lastly, we benchmarked different design choices for communication between shards and broadcast processors. We evaluated four configurations: running shards and broadcast processors in goroutines using channels or only using mutexes to handle concurrency. To test these choices, we implement the shard and broadcast processor by replacing the logic with a brief sleep to isolate the impact of using mutexes versus channels and goroutines. Benchmarks were run using Go's built-in tool [71] on a machine with an Intel Core i7 8700K CPU and 32GB RAM. Tests were named based on the concurrency mechanisms used, such as Lock-Lock (LL) for using locks for both shard and broadcast processor. The benchmarks simulate loads resembling those in Paxos based on preliminary testing. This includes processing 10,000 unique broadcast requests, each comprising 20 messages. These tests were run for all design alternatives: Lock-Lock, Lock-Channel, Channel-Lock, and Channel-Channel.

7.2 Experimental Results

Table 7.2 shows the latency of requests under normal operation. The data indicates that the systems are not overloaded and can process requests quickly. Both data-centric versions of the Paxos protocol perform similarly, suggesting that the overhead incurred by the broadcast framework is negligible under normal operations. The mean and median times of the Paxos implementation using the broadcast framework are comparable to those of the data-centric versions. However, the variance and maximum latency are higher, likely due to the client-side server implementation. The nodes need to establish a connection to the client-side server, which incurs overhead and extra latency during connection establishment. This trend is also evident in the PBFT implementations, where both versions create client connections when sending a reply. Despite this, the broadcast framework offers some optimizations, such as sharding, that make it slightly faster than the plain version. Although both versions use a single lock for the message log, the Gorums version benefits from parallelization techniques implemented in the broadcast framework, such as handling broadcast requests entirely asynchronously and separately.

Implementation	**Mean**	**Median**	**Stdev**	**Min**	**Max**
Paxos BC	1.93	1.53	1.94	0.89	23.21
Paxos QC	2.08	1.83	0.86	1.05	7.14
Paxos QCB	2.16	1.93	0.81	1.13	6.49
PBFT Plain	5.06	4.65	1.86	1.97	13.52
PBFT Gorums	3.30	2.66	2.09	1.49	23.44

Table 7.2: BC = BroadcastCall, QC = QuorumCall, QCB = QuorumCall using the broadcast framework. Ten clients send requests to the servers. Numbers are given in ms and represent the time taken from the moment the request is sent from a client until a response is received by the client.

Fig. 7.1 shows the throughput versus latency graph for the two implementations of PBFT. As the throughput increases, so does the latency, indicating linearity in the graphs. This linearity results from the use of a single lock when appending messages to the log, causing messages to wait for one another. Additionally, it is evident that both implementations could be further optimized, as they can only handle about 6,000 requests per second.

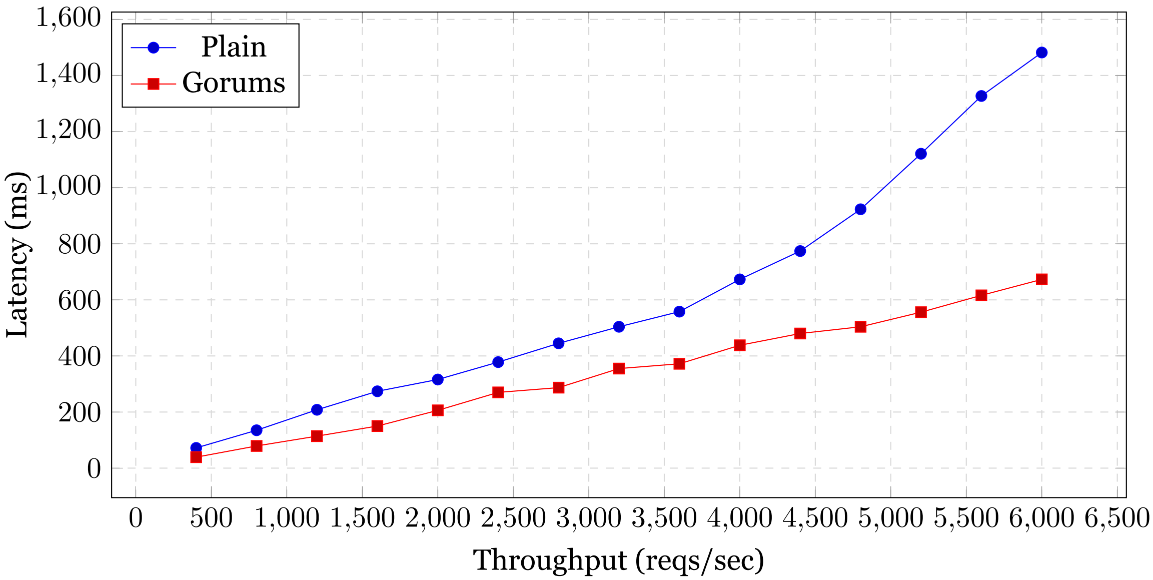


Figure 7.1: PBFT implemented using Gorums (Gorums) and PBFT implemented without using Gorums (Plain).

The Paxos implementations have very similar performances, as shown in Fig. 7.2. The process-centric version using BroadcastCall achieves higher throughput, likely due to not requiring the commit phase that its data-centric counterpart does. The data-centric Paxos implementation using the broadcast framework performs slightly worse when the number of client requests exceeds the maximum throughput of the servers. This decline in performance may be attributed to the overhead of creating a broadcast processor for each broadcast request. However, this overhead is almost negligible when the number of client requests is below the maximum throughput of the servers.

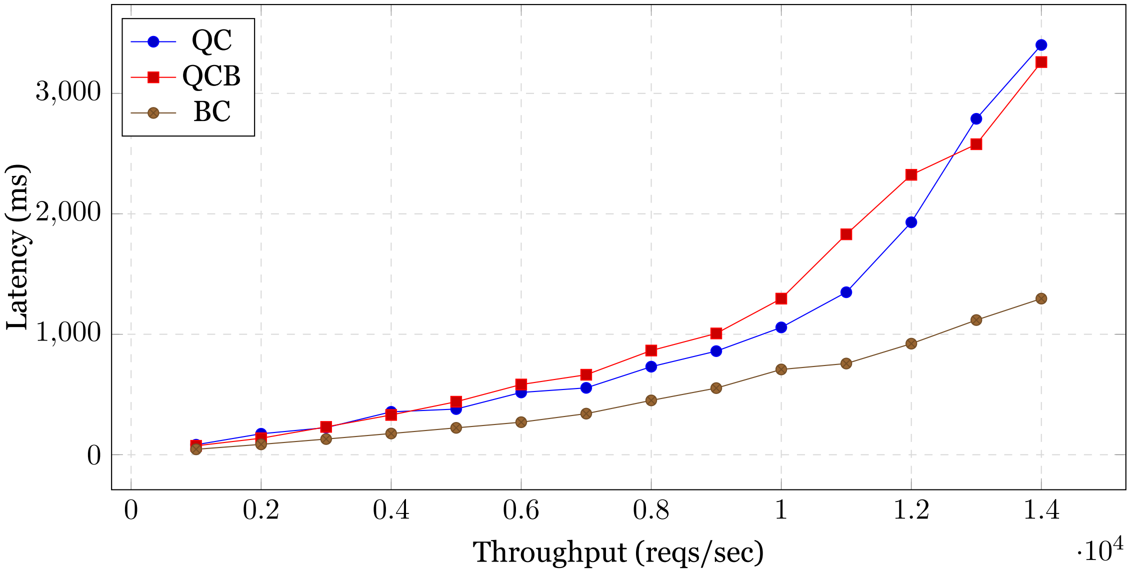


Figure 7.2: Paxos using QuorumCall (QC), QuroumCall with BroadcastOption (QCB), and BroadcastCall (BC).

The histograms in Fig. 7.3 and Fig. 7.4 show the latency distribution of the Paxos and PBFT implementations, respectively, close to their maximum throughput. These histograms corroborate the results in Table 7.2, demonstrating that the broadcast framework exhibits similar latencies to the baselines and does not negatively affect performance.

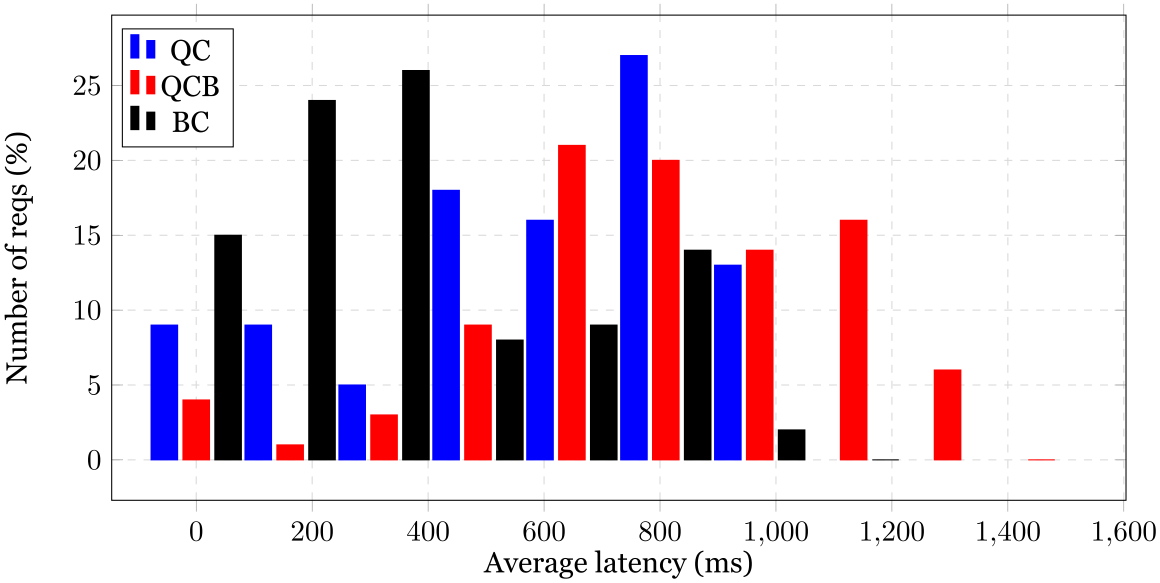


Figure 7.3: Performance comparison histogram displaying the three implementations of Paxos. The average latency is given in ms.

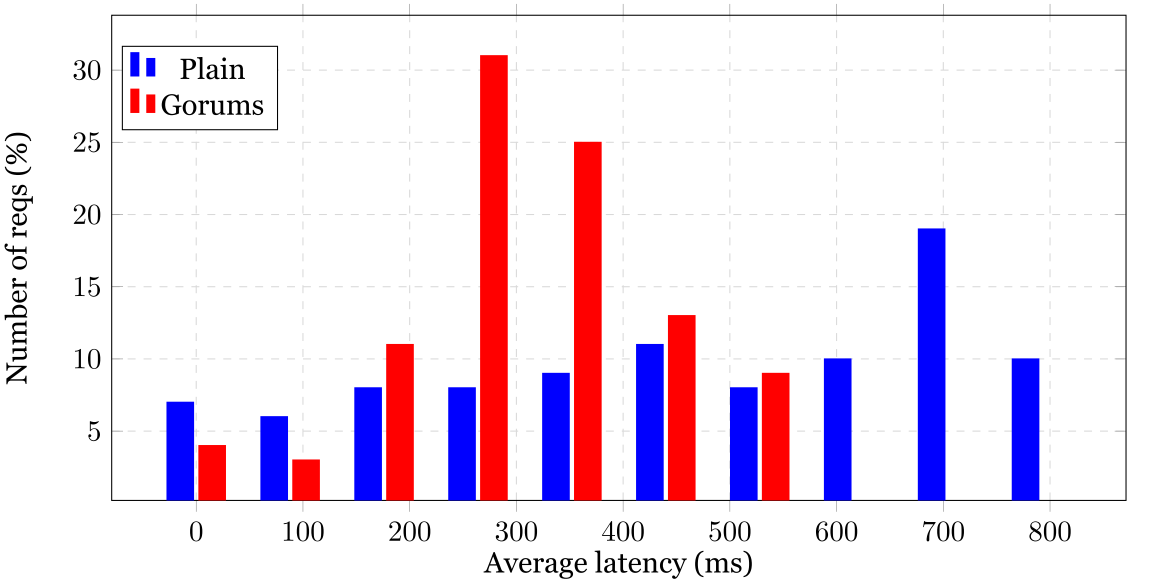


Figure 7.4: Performance comparison histogram displaying both implementations of PBFT. The average latency is given in ms.

There is a trade-off between writing performant, optimized code and maintaining clean, readable code. This project focuses on achieving high performance without compromising readability. Go's concurrency model, based on channels and goroutines [39], enables clear execution paths and data flows, enhancing code readability by adhering to language design patterns. The Go standard library's sync package [72] provides traditional locking mechanisms. The broadcast framework relies heavily on cached data like client connections and broadcast requests, which are well-suited for mutexes [72]. On the other hand, messages from clients or servers are passed to their respective broadcast processors using channels. While channels in Go also use mutexes, the tests conducted aim to determine if Go's runtime scheduler can optimize communication when using channels [73]. Table 7.3 indicates a performance edge when using mutexes for shards and channels for broadcast requests. Although using channels for both shards and broadcast requests is more idiomatic and readable [72], it incurs a high performance penalty. Therefore, mutexes are used as the concurrency mechanism for the shards.

Metric	**CC**	**LC**	**CL**	**LL**
sec/op (ms)	$196.1 \pm 2\%$	$67.62 \pm 1\%$	$263.8 \pm 2\%$	$70.73 \pm 0\%$
B/op (MiB)	$37.91 \pm 1\%$	$35.12 \pm 0\%$	$67.61 \pm 2\%$	$38.18 \pm 0\%$
allocs/op	$630.7K \pm 1\%$	$600.3K \pm 0\%$	$1.055M \pm 1\%$	$800.4K \pm 0\%$

Table 7.3: Benchmarks run using the built-in benchmarking tool in Go [71]. Tests are run on Intel Core i7 8700K CPU and 32GB RAM.

Chapter 8

Conclusions

8.1 Conclusion

In this thesis, we have examined and extended Gorums [16] to enable all-to-all communication in quorum-based systems. The objectives were to determine whether it was possible to integrate all-to-all communication for quorum-based systems with an easy-to-use API and to evaluate whether the framework would incur too much overhead when reliability and security were prioritized.

We began with an extensive analysis identifying the challenges of implementing abstractions for all-to-all communication. This analysis covered practical systems using all-to-all communication, various communication patterns, issues related to out-of-order messages, message duplication, cancellations, request lifetimes, broadcasting context, and authentication. We then developed a module specification for the framework, aiming to make it as extensible and flexible as possible. The analysis revealed that many algorithms share a common ground for the all-to-all communication pattern, making it feasible to create an abstraction for it. However, some tweaking is required for individual algorithms, so the abstraction must support customization to accommodate various protocols.

We demonstrated that all-to-all communication could be integrated with the Gorums framework by creating an abstraction named BroadcastCall. This is a transparent RPC invocation on a group of servers using the all-to-all communication pattern in all or parts of the implementation, addressing some of the problems related to distributed systems. The BroadcastCall uses semantics similar to the QuorumCall, making the API user-friendly for those with prior knowledge of Gorums. Furthermore, we introduced a broadcast struct in the server handlers, which contains most of the functionality related to the broadcast framework. Concentrating most of the functionality in this struct helps make the broadcast framework easy to use.

Using the metadata to relate messages to broadcast requests made it possible to enable the all-to-all communication pattern. We also introduced a version of Snowflake IDs [63] and sharding [65] to speed up processing and facilitate parallel processing. Additionally, the client-side server was crucial for enabling all-to-all communication, despite imposing extra overhead and processing on the client side. Given the constraints, it was not possible to accept replies from servers in other ways.

The broadcast framework is feature-rich, providing options for structured logging, authentication, execution ordering, connections between nodes, identification, broadcasting, and broadcast server configuration, making it extensible and flexible. These options enable the broadcast framework to facilitate protocols such as PBFT [20], Paxos [30], various broadcasting and gossiping protocols [11], and all-to-all specific consensus protocols used in blockchains [58][59]. Additionally, we implemented simple cancellation mechanisms, which can help improve performance in tail-tolerant systems. Authentication was also integrated to enhance the security of BroadcastCalls. Although several attack vectors exist, most can be mitigated using the authentication options provided by the broadcast framework. However, some attack surfaces related to rate limiting still need to be addressed, which we leave as further work.

Experimental evaluation shows that the broadcast framework can perform on par with baseline implementations of Paxos and PBFT. We implemented both process-centric and data-centric versions of Paxos and an implementation of PBFT using the broadcast framework. The experiments demonstrate that the broadcast framework provides good performance compared to baseline implementations without incurring excessive overhead. Compared to a simple, naive implementation using a single lock, the broadcast framework outperformed the baseline PBFT implementation.

8.2 Limitations and Further Work

This section provides insights into the limitations of the broadcast framework and proposes areas for future work related to the broadcast framework presented in this thesis.

Evaluation: The experiments used in the evaluation are limited and do not cover all aspects of the new broadcast framework. Hence, more tests, experiments, and benchmarks should be performed against state-of-the-art implementations of PBFT [20] and Paxos [30] as well as various other algorithms. It would also be beneficial to test the broadcast framework in adversarial scenarios, such as failing nodes, slow nodes, network partitions, and different server configurations with varying memory sizes, CPU speeds, and numbers of servers.

Ordering: We examined issues related to the ordering of messages but decided not to implement specific ordering techniques. Instead, we implemented a solution for setting the execution order of handlers. As further work, we could implement different ordering strategies such as FIFO, total order, and causal order [11].

Batching: Batching is an optimization strategy that can significantly improve performance in certain scenarios [74]. Implementing batching using the broadcast framework can be challenging because the framework itself handles the routing of messages. As further work, we could include batching as an option, which would also route client requests automatically after a batch is processed by the servers.

All-to-all Personalized Broadcast: Currently, a client sends an identical broadcast request to all servers. However, we could add support for custom requests to each server, facilitating all-to-all personalized broadcasts.

Testing: Concurrency bugs can be hard to spot and debug; thus, integrating tooling such as Gobench [18] to test the user application for concurrency bugs would be beneficial.

Rate Limiting: Denial-of-service attacks can be executed on the broadcast framework by sending an extensive number of requests. Therefore, implementing rate-limiting options could be crucial in production systems, particularly those exposed to the public [75].

Better Error Handling: The current version of the broadcast framework does not support servers replying with an error to the client. Presently, errors are ignored because it is not possible to implicitly deduce how many replies are expected in a BroadcastCall. This is due to the client's ability to send a broadcast request to only a subset of servers. Therefore, implementing error handling would require changing the signature of the quorum function to accept a slice of errors, breaking the uniform semantics used in both QuorumCall and BroadcastCall. Evaluating this change is an area of future work, as providing good abstractions for handling errors could be highly beneficial.

Extensibility: The focus of this thesis has been on quorum-based systems utilizing all-to-all communication. However, it remains to be evaluated if Gorums can be effectively applied to other types of systems, such as Apache Spark [29], Kubernetes[2], and others beyond quorum-based systems.

Several Views: Currently, a server can only have a single view for communication. However, some implementations or configurations could benefit from supporting multiple views on each server. For example, using one view for acceptors and another for learners in Paxos.

Optimization: The focus of this thesis has been on supporting the all-to-all communication pattern, so the metadata has not been optimized. For example, IPv4 addresses could be represented as 4 bytes instead of UTF-8 encoded strings [76], and method names could be represented as unsigned integers instead of strings. Furthermore, we could explore the possibility of serializing messages once when broadcasting, instead of serializing messages for every node.

Middleware: The broadcast framework performs some processing before a message is handled by the server implementation provided by the user. Therefore, it could be beneficial for users to specify middleware that runs immediately upon message arrival. This middleware could be used for tasks such as validation or logging. Providing this functionality as an option would allow users to specify middleware for each handler, enhancing flexibility and control.

Data Structures: The broadcast framework could support creating internal data structures to speed up operations, such as storing to a map and creating an optimized list for concurrency. This would be useful in PBFT for the message log. Such enhancements could significantly improve performance compared to using a single lock. Additionally, it would mitigate the risk of concurrency bugs related to checking whether a variable exists before adding an element. If two handlers run concurrently, we could end up adding an element to the list in both handlers if the checking and adding operation is not protected by the same lock at the same time.

Server-Side Quorum Functions: Each server handler in the broadcast framework currently accepts a single request message. This could be extended to function similarly to the quorum function, where all currently received messages are provided to the server handler each time a new message arrives. Such an extension could simplify the logic in the server handlers, allowing the processing to be done once instead of multiple times. This feature could be provided as an option, as it may introduce some overhead due to the need for caching messages. Furthermore, it would enhance the security of the broadcast framework by enabling the use of a quorum of messages to determine the broadcast metadata in case of a Byzantine node.

Type Safety: Using SendToClient() as the sole method for client replies has its advantages and disadvantages. It simplifies the code by reducing methods to a single unified call, enhancing readability in complex branching scenarios. However, it compromises type safety by accepting any proto message, potentially causing the application to panic if the wrong message type is provided. Additionally, SendToClient() does not clarify which execution path it concludes, whereas method names like Path1Reply() and Path2Reply() could improve readability. The current implementation's naming convention avoids ambiguity about the reply recipient, as servers do not receive replies like clients. Improving the server's ability to initiate requests and receive replies is an area for further work that could be evaluated. Therefore, a logical next step would be to provide type safety when sending replies.

List of Figures

List of Tables

List of Algorithms

Listings

Bibliography

[1] C. Kidd, *Distributed Systems Explained | Splunk*. [Online]. Available: https://www.splunk.com/en_us/blog/learn/distributed-systems.html (visited on 05/27/2024).

[2] *Clustering and HA | KubeMQ Docs*, en, Dec. 2022. [Online]. Available: https://docs.kubemq.io/learn/cluster-scale (visited on 05/27/2024).

[3] *Raft consensus in swarm mode*, en, 0. [Online]. Available: https://docs.docker.com/engine/swarm/raft/ (visited on 05/27/2024).

[4] *Apache Kafka*, en. [Online]. Available: https://kafka.apache.org/ (visited on 06/04/2024).

[5] *Quorum Queues | RabbitMQ*, en. [Online]. Available: https://www.rabbitmq.com/docs/quorum-queues (visited on 05/27/2024).

[6] spelluru, *Introduction to Azure Service Bus, an enterprise message broker - Azure Service Bus*, en-us, Feb. 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview (visited on 06/04/2024).

[7] *What is Amazon Simple Queue Service - Amazon Simple Queue Service*. [Online]. Available: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html (visited on 05/27/2024).

[8] *Kubernetes Components*, en, Section: docs. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/ (visited on 05/27/2024).

[9] S. Pedersen, H. Meling, and L. Jehl, “An Analysis of Quorum-based Abstractions: A Case Study using Gorums to Implement Raft,” Jul. 2018, pp. 29–35. DOI: 10.1145/3231104.3231957.

[10] X. Défago, A. Schiper, and P. Urbán, "Totally Ordered Broadcast and Multicast Algorithms: A Comprehensive Survey," Jan. 2000.

[11] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, en. Berlin, Heidelberg: Springer, 2011, ISBN: 978-3-642-15259-7 978-3-642-15260-3. DOI: `10.1007/978-3-642-15260-3`. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-15260-3` (visited on 05/25/2024).

[12] P. Kuznetsov and A. Tonkikh, *Asynchronous Reconfiguration with Byzantine Failures*, arXiv:2005.13499 [cs], Oct. 2021. DOI: `10.48550/arXiv.2005.13499`. [Online]. Available: `http://arxiv.org/abs/2005.13499` (visited on 05/27/2024).

[13] X. Luo, W. Shen, S. Mu, and T. Xu, "DepFast: Orchestrating code of quorum systems," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 557–574, ISBN: 978-1-939133-29-14. [Online]. Available: `https://www.usenix.org/conference/atc22/presentation/luo`.

[14] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: Language support for building distributed systems," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 179–188, Jun. 2007, ISSN: 0362-1340. DOI: `10.1145/1273442.1250755`. [Online]. Available: `https://doi.org/10.1145/1273442.1250755` (visited on 05/27/2024).

[15] J. Gabrielson, *Challenges with distributed systems*, en-US. [Online]. Available: `https://aws.amazon.com/builders-library/challenges-with-distributed-systems/` (visited on 05/27/2024).

[16] T. E. Lea, L. Jehl, and H. Meling, "Towards New Abstractions for Implementing Quorum-Based Systems," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, ISSN: 1063-6927, Jun. 2017, pp. 2380–2385. DOI: `10.1109/ICDCS.2017.166`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/7980198` (visited on 05/25/2024).

[17] *Documentation - The Go Programming Language*, en. [Online]. Available: `https://go.dev/doc/` (visited on 05/25/2024).

[18] T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue, "GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2021, pp. 187–199. DOI: 10.1109/CGO51591.2021.9370317. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9370317 (visited on 05/26/2024).

[19] M. Chabbi and M. K. Ramanathan, "A study of real-world data races in Golang," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022, New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 474–489, ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523720. [Online]. Available: https://doi.org/10.1145/3519939.3523720 (visited on 06/12/2024).

[20] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," en, in *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*, New Orleans, LA: USENIX Association, Feb. 1999. [Online]. Available: https://www.usenix.org/conference/osdi-99/practical-byzantine-fault-tolerance.

[21] P. Tennage, C. Basescu, L. Kokoris-Kogias, *et al.*, "QuePaxa: Escaping the tyranny of timeouts in consensus," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23, New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 281–297, ISBN: 9798400702297. DOI: 10.1145/3600006.3613150. [Online]. Available: https://dl.acm.org/doi/10.1145/3600006.3613150 (visited on 05/27/2024).

[22] E. Zamora-Gómez, P. García-López, and R. Mondéjar, "Continuation Complexity: A Callback Hell for Distributed Systems," en, in *Euro-Par 2015: Parallel Processing Workshops*, S. Hunold, A. Costan, D. Giménez, *et al.*, Eds., Cham: Springer International Publishing, 2015, pp. 286–298, ISBN: 978-3-319-27308-2. DOI: 10.1007/978-3-319-27308-2_24.

[23] H. Meling, A. Montresor, B. E. Helvik, and O. Babaoglu, "Jgroup/ARM: A distributed object group platform with autonomous replication management," en, *Software: Practice and Experience*, vol. 38, no. 9, pp. 885–923, 2008, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.853, ISSN: 1097-024X. DOI: 10.1002/spe.853. [Online]. Available: https:

//onlinelibrary.wiley.com/doi/abs/10.1002/spe.853 (visited on 05/27/2024).

[24] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, “All about Eve: {Execute-Verify} Replication for {Multi-Core} Servers,” en, 2012, pp. 237–250, ISBN: 978-1-931971-96-6. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kapritsos (visited on 05/27/2024).

[25] A. Massini, “All-to-all personalized communication on multistage interconnection networks,” *Discrete Applied Mathematics*, vol. 128, no. 2, pp. 435–446, Jun. 2003, ISSN: 0166-218X. DOI: 10.1016/S0166-218X(02)00504-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166218X02005048 (visited on 05/25/2024).

[26] BianchiniMonica, GoriMarco, and ScarselliFranco, “Inside PageRank,” EN, *ACM Transactions on Internet Technology (TOIT)*, Feb. 2005, Publisher: ACMPUB27New York, NY, USA. DOI: 10.1145/1052934.1052938. [Online]. Available: https://dl.acm.org/doi/10.1145/1052934.1052938 (visited on 06/04/2024).

[27] *Graph Partition - an overview | ScienceDirect Topics*. [Online]. Available: https://www.sciencedirect.com/topics/mathematics/graph-partition (visited on 06/04/2024).

[28] R. Rocha and F. Silva, “Parallel Sorting Algorithms,” en, [Online]. Available: https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_sorting.pdf.

[29] *Apache Spark™ - Unified Engine for large-scale data analytics*. [Online]. Available: https://spark.apache.org/ (visited on 06/04/2024).

[30] L. Lamport, “Paxos made simple,” *ACM Sigact News*, vol. 32, no. 4, pp. 51–58, 2001. [Online]. Available: https://lamport.azurewebsites.net/pubs/paxos-simple.pdf (visited on 05/26/2024).

[31] *Apache Spark: Core concepts, architecture and internals*, en-us. [Online]. Available: https://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/ (visited on 05/27/2024).

[32] O. Alpos, C. Cachin, B. Tackmann, and L. Zanolini, *Asymmetric Distributed Trust*, arXiv:1906.09314 [cs], May 2024. DOI: `10.48550/arXiv.1906.09314`. [Online]. Available: `http://arxiv.org/abs/1906.09314` (visited on 05/27/2024).

[33] C. Cachin and L. Zanolini, *From Symmetric to Asymmetric Asynchronous Byzantine Consensus*, arXiv:2005.08795 [cs], Jun. 2021. DOI: `10.48550/arXiv.2005.08795`. [Online]. Available: `http://arxiv.org/abs/2005.08795` (visited on 05/27/2024).

[34] *Global Payments & Financial Solutions for Businesses | Ripple*, en. [Online]. Available: `https://ripple.com` (visited on 05/27/2024).

[35] *Stellar | A Blockchain Network for Payments and Tokenization*, en. [Online]. Available: `https://stellar.org` (visited on 05/27/2024).

[36] *Group Communication in distributed Systems*, en-US, Section: Distributed System, Dec. 2020. [Online]. Available: `https://www.geeksforgeeks.org/group-communication-in-distributed-systems/` (visited on 05/26/2024).

[37] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," *ACM Computing Surveys*, vol. 33, no. 4, pp. 427–469, Dec. 2001, ISSN: 0360-0300. DOI: `10.1145/503112.503113`. [Online]. Available: `https://doi.org/10.1145/503112.503113` (visited on 05/27/2024).

[38] *gRPC*, en. [Online]. Available: `https://grpc.io/` (visited on 05/25/2024).

[39] O. Özşahin, *Concurrency with Goroutines and Channels in Go*, en, Sep. 2023. [Online]. Available: `https://okanexe.medium.com/concurrency-with-goroutines-and-channels-in-go-b6e8bace6d94` (visited on 05/25/2024).

[40] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978, ISSN: 0001-0782. DOI: `10.1145/359576.359585`. [Online]. Available: `https://dl.acm.org/doi/10.1145/359576.359585` (visited on 05/26/2024).

[41] *Grpc-go/backoff.go at v1.64.0 · grpc/grpc-go*, en. [Online]. Available: `https://github.com/grpc/grpc-go/blob/v1.64.0/backoff.go` (visited on 06/04/2024).

[42] *Protocol Buffers*, en. [Online]. Available: https://protobuf.dev/ (visited on 05/26/2024).

[43] *Protocol Buffer Compiler Installation*, en, Section: docs. [Online]. Available: https://grpc.io/docs/protoc-installation/ (visited on 05/26/2024).

[44] K. Ngo, S. Sen, and W. Lloyd, "Tolerating slowdowns in replicated state machines using copilots," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 583–598, ISBN: 978-1-939133-19-9. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/ngo.

[45] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002, ISSN: 0734-2071. DOI: 10.1145/571637.571640. [Online]. Available: https://doi.org/10.1145/571637.571640 (visited on 05/26/2024).

[46] T. C. Frausing, "Reconfiguration Abstractions for Gorums," eng, Accepted: 2018-09-25T12:15:20Z, M.S. thesis, University of Stavanger, Norway, Jun. 2018. [Online]. Available: https://uis.brage.unit.no/uis-xmlui/handle/11250/2564390 (visited on 06/11/2024).

[47] E. Gafni and L. Lamport, "Disk Paxos," en, *Distributed Computing*, vol. 16, no. 1, pp. 1–20, Feb. 2003, ISSN: 1432-0452. DOI: 10.1007/s00446-002-0070-8. [Online]. Available: https://doi.org/10.1007/s00446-002-0070-8 (visited on 06/11/2024).

[48] G. Chockler and D. Malkhi, "Active disk paxos with infinitely many processes," in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, ser. PODC '02, New York, NY, USA: Association for Computing Machinery, Jul. 2002, pp. 78–87, ISBN: 978-1-58113-485-8. DOI: 10.1145/571825.571837. [Online]. Available: https://doi.org/10.1145/571825.571837 (visited on 06/11/2024).

[49] *What is MapReduce?* en-US, Dec. 2021. [Online]. Available: https://www.databricks.com/glossary/mapreduce (visited on 06/12/2024).

[50] *Scatter-Gather | MuleSoft Documentation*. [Online]. Available: https://docs.mulesoft.com/mule-runtime/3.9/scatter-gather (visited on 06/12/2024).

[51] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence.," Jan. 1973, pp. 235–245.

[52] *Build, Operate, and Secure Low Latency Systems*. [Online]. Available: `https://akka.io/` (visited on 06/12/2024).

[53] *How the Actor Model Meets the Needs of Modern, Distributed Systems • Akka Documentation*. [Online]. Available: `https://doc.akka.io/docs/akka/current/typed/guide/actors-intro.html` (visited on 06/12/2024).

[54] *Index - Erlang/OTP*, en. [Online]. Available: `https://www.erlang.org/` (visited on 06/13/2024).

[55] A. D. Meulemeester, *Anthdm/hollywood*, original-date: 2023-01-27T14:36:30Z, Jun. 2024. [Online]. Available: `https://github.com/anthdm/hollywood` (visited on 06/12/2024).

[56] *Open MPI: Open Source High Performance Computing*. [Online]. Available: `https://www.open-mpi.org/` (visited on 06/12/2024).

[57] *17.2.16. MPI_alltoall — Open MPI 5.0.x documentation*. [Online]. Available: `https://docs.open-mpi.org/en/v5.0.x/man-openmpi/man3/MPI_Alltoall.3.html` (visited on 06/12/2024).

[58] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018. arXiv: `1807.04938`. [Online]. Available: `http://arxiv.org/abs/1807.04938`.

[59] *Stellar Consensus Protocol (SCP) | Stellar Docs*, en, Mar. 2024. [Online]. Available: `https://developers.stellar.org/docs/learn/fundamentals/stellar-consensus-protocol` (visited on 06/05/2024).

[60] *Deadlines*, en, Section: docs. [Online]. Available: `https://grpc.io/docs/guides/deadlines/` (visited on 06/09/2024).

[61] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013, ISSN: 0001-0782. DOI: `10.1145/2408776.2408794`. [Online]. Available: `https://dl.acm.org/doi/10.1145/2408776.2408794` (visited on 05/27/2024).

[62] *Basics tutorial*, en, Section: docs. [Online]. Available: `https://grpc.io/docs/languages/go/basics/` (visited on 05/26/2024).

[63] M. Jarmółkiewicz, *The Unique Features of Snowflake ID and its Comparison to UUID*, en. [Online]. Available: `https://softwaremind.com/blog/the-unique-features-of-snowflake-id-and-its-comparison-to-uuid/` (visited on 05/26/2024).

[64] *Uuid package - github.com/google/uuid - Go Packages*. [Online]. Available: `https://pkg.go.dev/github.com/google/uuid` (visited on 05/26/2024).

[65] *What is Sharding? - Database Sharding Explained - AWS*. [Online]. Available: `https://aws.amazon.com/what-is/database-sharding/` (visited on 05/26/2024).

[66] C. C. Editor, *Transmission Control Protocol - Glossary | CSRC*, EN-US. [Online]. Available: `https://csrc.nist.gov/glossary/term/transmission_control_protocol` (visited on 05/26/2024).

[67] *Authentication*, en, Section: docs. [Online]. Available: `https://grpc.io/docs/guides/auth/` (visited on 06/11/2024).

[68] *Cloud Computing Services | Microsoft Azure*, en-US. [Online]. Available: `https://azure.microsoft.com/en-us` (visited on 06/06/2024).

[69] anaharris, *What are Azure availability zones?* en-us, Mar. 2024. [Online]. Available: `https://learn.microsoft.com/en-us/azure/reliability/availability-zones-overview` (visited on 06/06/2024).

[70] *Pricing - Linux Virtual Machines | Microsoft Azure*, en. [Online]. Available: `https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/` (visited on 06/06/2024).

[71] *Testing package - testing - Go Packages*. [Online]. Available: `https://pkg.go.dev/testing` (visited on 06/06/2024).

[72] *Go Wiki: Use a sync.Mutex or a channel? - The Go Programming Language*, en. [Online]. Available: `https://go.dev/wiki/MutexOrChannel` (visited on 05/25/2024).

[73] ivanspasov99, *Mastering Concurrency: Unveiling the Magic of Go's Scheduler*, en, Section: Additional Blogs by SAP, Nov. 2023. [Online]. Available: `https://community.sap.com/t5/additional-blogs-by-sap/`

mastering-concurrency-unveiling-the-magic-of-go-s-scheduler/ba-p/13577437 (visited on 05/26/2024).

[74] N. Santos and A. Schiper, “Tuning Paxos for High-Throughput with Batching and Pipelining,” Jan. 2012, ISBN: 978-3-642-25958-6. DOI: 10.1007/978-3-642-25959-3_11.

[75] R. Yogesh Patil and L. Ragha, “A rate limiting mechanism for defending against flooding based distributed denial of service attack,” in *2011 World Congress on Information and Communication Technologies*, Dec. 2011, pp. 182–186. DOI: 10.1109/WICT.2011.6141240. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6141240 (visited on 06/14/2024).

[76] *Encoding*, en, Section: programming-guides. [Online]. Available: https://protobuf.dev/programming-guides/encoding/ (visited on 06/14/2024).

Appendix A

Attachments

This appendix contains all the code developed and used in this thesis. The code can be found in the following repository or in the embedded zip file:

- **Repository**: `https://github.com/aleksander-vedvik/Master`
- **Zip file**: code.zip

The repository contains the benchmarking code, the Paxos and PBFT implementations used in the evaluation, the example used in the thesis, and the complete codebase of Gorums, including the broadcasting framework. It also contains instructions on how to run the benchmarks.

The Gorums repository is open-source, and the broadcasting framework developed in this thesis has been integrated into a branch on the Gorums repository. All changes made are available in the draft pull request added to the GitHub repository. For access to the full code, including the broadcasting framework, please refer to the following GitHub repository:

- **Gorums repository**: `https://github.com/relab/gorums`
- **Branch**: `https://github.com/relab/gorums/tree/multiparty`
- **Draft pull request**: `https://github.com/relab/gorums/pull/176`

University
of Stavanger

4036 Stavanger
Tel: +47 51 83 10 00
E-mail: post@uis.no
www.uis.no

Cover Photo: Hein Meling