University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Computer Science	Spring semester, 2018 Open / ~~Confidential~~
Author: Tor Christian Frausing	 ………………………………………… (signature of author)

Programme coordinator: Hein Meling

Supervisor(s): Hein Meling and Leander Jehl

Title of Master's Thesis:
Reconfiguration Abstractions for Gorums
Providing Out-of-the-box Reconfiguration to Systems Using the Gorums Framework

Credits: 30 ECTS

Keywords: Distributed Systems • Reconfiguration • Dynamic Systems • Gorums • Consensus • Atomic Storage • State Machine Replication	Number of pages: 123 + supplemental material/other: - Code included in PDF - Experimental data included in PDF Stavanger, June 15, 2018

Reconfiguration Abstractions for Gorums

Providing Out-of-the-box Reconfiguration to Systems Using the Gorums Framework

Tor Christian Frausing

15 June 2018

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

Abstract

To provide the high availability expected of a modern data service, the service needs to be capable of adapting to a multitude of scenarios, e.g., increased demand and unexpected failures. A common technique for mitigating these issues and to improve upon the service's fault-tolerance, is to replicate the service to a set of independent machines. Gorums is a novel *remote procedure call* (RPC) framework designed for alleviating developers of the complexity associated with building distributed fault-tolerant services. Machines have a tendency to eventually fail. Therefore, to ensure continued availability of a service and to retain its fault-tolerance, we need to be able to replace faulty machines without disrupting the user's experience through reconfiguration.

This thesis examines and extends the Gorums framework with easy to use, available, and adaptable reconfiguration abstractions. By providing a generalized out-of-the-box reconfiguration component capable of introducing reconfiguration capabilities to arbitrary services, the extended Gorums framework is able to further alleviate developers of the complex, tedious, and time-consuming processes of building truly fault-tolerant services. The reconfiguration component masks the intricacies of reconfiguration and instead presents developers with an easy to use and intuitive reconfiguration API. An experimental evaluation of the extended Gorums framework and its reconfiguration component is also presented.

Acknowledgements

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

I would also like to express my gratitude toward my co-supervisor, Leander Jehl, for his dedication and contribution to my work. The weekly meetings, continued discussions, and insights have all been very helpful in pointing me in the right direction.

Table of Contents

Abstract **i**

Acknowledgements **ii**

Table of Contents **iii**

1 Introduction **1**

1.1 Contributions and Outline 3

2 Background **6**

2.1 Consensus 6

2.2 Reconfiguration 8

2.3 The Paxos Protocol 11

3 Gorums **17**

3.1 Quorum Calls 17

3.2 Quorum Functions 18

3.3 Extensions 21

3.4 Architecture and Implementation Details 24

4 Reconfiguration Techniques **27**

4.1 State Transfer 27

4.2 Consensus-based Approaches 31

4.3 Consensus-free Approaches . . . 34

5 Design and Implementation **37**
5.1 Design . . . 37
5.1.1 Reconfiguration Scheme . . . 37
5.1.2 Consensus-based Approach . . . 44
5.1.3 Consensus-free Approach . . . 53
5.2 Implementation . . . 61
5.2.1 System Overview and Architecture . . . 62
5.2.2 Quorum Specification . . . 65
5.2.3 Alternative Approach . . . 67

6 Practical Applications **70**
6.1 Atomic Register . . . 70
6.2 Vertical Paxos . . . 76

7 Experimental Evaluation **85**
7.1 Experimental Setup . . . 85
7.2 Common Case Operations . . . 87
7.3 Reconfiguration Overhead . . . 89

8 Conclusion and Further Work **96**
8.1 Conclusion . . . 96
8.2 Further Work . . . 98

A Attachments **i**
A.1 Source Code . . . i
A.2 Printer-Friendly Version of Thesis . . . i
A.3 Raw Experimental Data . . . ii

B Complete Experimental Data **iii**

List of Algorithms **v**

List of Figures **vi**

List of Listings **vii**

List of Tables **viii**

Bibliography **ix**

1 Introduction

To provide the high availability expected of a modern data service, the service needs to be capable of adapting to a multitude of scenarios. Increasing demand, system maintenance, and unexpected failures are all scenarios that need to be accounted for. These scenarios can all disrupt the service, leaving it in a state where it is incapable of servicing all of the incoming requests and is, as a consequence, perceived as unavailable from a user's point of view. A common technique for mitigating these issues and to improve upon the service's resilience, is to replicate the service to a set of independent machines [8, 55].

By replicating the service to a set of machines, we increase the service's fault-tolerance, enabling it to support up to f failures without disrupting its operation. Another advantage is that of greater flexibility. Originally, all incoming requests were served by a single machine running the service, whereas after replication, incoming requests can be uniformly distributed to the set of machines, balancing the workload and increasing the service's robustness to unexpected spikes. However, depending on the nature of the service, replicating its logic may be a difficult and complex undertaking. The problem of ensuring data consistency in a replicated setting is often nontrivial, and has accordingly been the topic of extensive research [6, 18, 41, 55]. In an effort to alleviate some of this complexity and ease the development of replicated and fault-tolerant services, Gorums [43, 44], a novel *remote procedure call* (RPC) [7] framework, provides powerful and flexible abstractions for interacting with a set of machines.

Nonetheless, machines have a tendency to eventually fail, and failed machines are no longer able to partake in the service's operations. If this problem is not addressed, the cumulative number of failed machines will eventually exceed the service's fault-tolerance f and leave the service in an unresponsive and unavailable state. Therefore, to ensure continued availability and to retain the service's fault-tolerance throughout its lifetime, we need to be able to replace faulty machines without disrupting the user's experience. Henceforth, the procedure of modifying the set is referred to as a reconfiguration. In addition to replacing faulty machines, reconfigurations can alter the characteristics of the service by adding or removing machines. For example, adding new machines to better accommodate for an increasing workload, support taking down a subset of machines for maintenance, or to increase the service's fault-tolerance.

Performing a reconfiguration without disrupting the service is an advanced and highly sophisticated procedure that introduces a new series of complications developers need to address. Hence, similarly to the replication of a service, extending the service with reconfiguration capabilities is a nontrivial and tedious undertaking that adds yet another layer of complexity. In this thesis, we alleviate developers of this complexity by incorporating new reconfiguration abstractions with the Gorums framework in an effort to provide a generalized out-of-the-box reconfiguration scheme for arbitrary services.

Even though reconfigurations are an essential and important component of ensuring continued availability and in maintaining the fault-tolerance of a service, its complexity and nontriviality has resulted in a common practice being to omit reconfigurations altogether during the initial development [16]. Adding reconfiguration capabilities upon a later revision, if adding it at all. Moreover, reconfigurations are only issued to counteract anomalies and unexpected failures. Hence, when considering the frequency of reconfiguration operations versus common case operations, it is evident that reconfigurations are infrequent and only constitute a tiny fraction of the requests processed by the service. Consequently, although it is very important to support reconfigurations, it is not necessarily important to optimize the procedure. For these reasons, we believe it is beneficial to produce a generalized out-of-the-box reconfiguration component in the Gorums framework. By providing reconfiguration capabilities to arbitrary services, we are able to relieve developers

of this complex, tedious, and time-consuming process.

The overall goal of this thesis is to examine and extend the Gorums framework with easy to use, available, and adaptable reconfiguration abstractions. To this end, we develop a reconfiguration scheme with design goals of (*i*) minimal performance overhead during common case operations, and (*ii*) providing sufficient adaptable abstractions for easily introducing reconfiguration to arbitrary services. As part of our reconfiguration scheme, we develop one consensus-based and one consensus-free reconfiguration technique that we evaluate against each other through experimental evaluation. While the former technique relies on the power of consensus to choose the reconfiguration target, the latter imposes a convergence property on configurations to circumvent consensus and is thus suitable for operating in asynchronous environments. Moreover, as a result of our examination of the current state of the framework, we propose two different approaches for incorporating our generalized reconfiguration scheme and evaluate these propositions against each other through experimental evaluation.

1.1 Contributions and Outline

We make the following contributions in this thesis:

- We analyze and adapt the Paxos protocol to the data-centric model. The adaption provides a consensus instance to an infinite amount of clients and enables clients to learn the chosen value in a single round trip, provided that the chosen value is decided.
- We systemize reconfiguration. Detailing requirements, available techniques, and common issues that need to be addressed. Then, we survey state-of-the-art reconfiguration algorithms, categorizing them based on their techniques and comparing them against each other and the new reconfiguration techniques designed as part of this thesis.
- We create a new reconfiguration component capable of transparently reconfiguring arbitrary services. To this end, we design and develop two new data-centric reconfiguration techniques for use by our component.

- We examine the current state of the Gorums framework and propose two possible methods for integrating the new reconfiguration component into the framework.
- We provide two practical applications, showcasing how our reconfiguration component can alleviate developers by reducing the complexity of building fault-tolerant services. In the first example we extend a data-centric distributed storage solution with reconfiguration capabilities. In the second case, we highlight how our data-centric reconfiguration component can replace the reconfiguration component in a process-centric distributed setting.
- We provide an implementation of the reconfiguration component and its reconfiguration techniques for both integration methods. The implementations are evaluated against each other, assessing each technique's and implementation's overhead on common case operations for both a non-reconfiguring environment and a reconfiguring environment.
- We make numerous additional contributions to the Gorums framework. Examples include adding support for altering the set of available nodes known to the framework during runtime and extend the quorum specification definition to include a factory method.

The remainder of this thesis is outlined as follows:

Chapter 2 introduces the relevant background material, focusing on the core concepts of consensus and reconfiguration. In addition, it details the design of our data-centric Paxos variant.

Chapter 3 provides a thorough introduction to the Gorums framework, including its abstractions, design, architecture, and available extensions.

Chapter 4 presents reconfiguration techniques. This includes the concept of state transfer and both consensus-based and consensus-free approaches for establishing the reconfiguration target. Lastly, state-of-the-art reconfiguration algorithms are classified and their techniques are compared against our generalized reconfiguration scheme.

Chapter 5 describes the design and implementation of our generalized reconfiguration scheme. Providing an in-depth description of its consensus-based and consensus-free reconfiguration techniques and our two approaches for integrating the scheme in the Gorums framework.

Chapter 6 presents two distributed algorithms, highlighting their integration with our reconfiguration scheme and how our reconfiguration scheme can either extend the algorithm with reconfiguration capabilities or replace the algorithm's reconfiguration component.

Chapter 7 provides an experimental evaluation of our reconfiguration scheme, both in terms of added cost to common case operations and the overhead incurred on operations that are concurrent to reconfiguration operations.

Chapter 8 concludes and presents suggestions for further work.

2
Background

This chapter provides a thorough introduction to core concepts and techniques used throughout this thesis. First we provide an in-depth presentation of the core-concepts: consensus and reconfiguration. Then, we present the Paxos protocol. While Paxos is not the primary focus of this thesis, it remains a fundamental building block for our work.

2.1 Consensus

Consensus is one of the most fundamental problems of distributed systems [9], and the consensus abstraction is defined as the process of agreeing upon a single value from a set of possible values. In a distributed setting, each process p from a set $\mathcal{P}$ of processes can propose some value to the consensus instance. Eventually, the consensus instance decides upon a single value from the set of proposed values, and every correct process in $\mathcal{P}$ learns of this chosen value. Due to the nature of distributed systems, an environment prone to network partitions and crash-faults, we require every process p to adhere to the following safety properties [38]:

- Only proposed values can be decided.
- Only a single value is decided.
- A process never learns that a value has been decided, unless it actually has been.

To illustrate the need for consensus in a typical distributed system, we assume a replicated state machine (RSM) [55]. A state machine is a deterministic protocol for keeping track of and updating a local state ϕ. By replicating a service to a set of state machines, we replicate the system's state to multiple processes such that the overall system provides a degree of fault-tolerance. Thus, the system is able to withstand a finite number of failures without affecting the availability of the service. To ensure that the state of different replicas remain consistent throughout the system's lifespan, updates to ϕ need to be executed in the same order at every replica. This is achieved by requiring a total order among all received requests (i.e., requests are synchronized), and accomplished by relying on the safety properties of a consensus instance. Consensus enforces a strict serializability on requests and can therefore guarantee that requests are invoked in the same order at different replicas. Further, similarly to other distributed primitives, consensus guarantees that every request is received by all correct replicas.

Requests to the system, received sequentially or concurrently, are proposed by the different replicas to to the consensus instance, and eventually it decides upon the next request to execute. Replicas that learn chosen requests, execute the requests in the order they were chosen. Thus, ensuring that ϕ is updated in a deterministic manner among all correct replicas. In theory, a consensus instance can only be used to decide upon a single value, how this is solved in practice is discussed further in Section 2.3.

Any algorithm that helps multiple processes maintain a common state or agree on future actions, in a model where processes are allowed to fail, is in fact solving a consensus problem [9]. For example, the techniques applied by the blockchain technology commonly found in use by modern cryptocurrencies solve a consensus problem in a new nontraditional manner [50]. However, for the timeless asynchronous system model, consensus has been shown to be impossible to solve even if only a single process fails by crashing [19]. Due to the fact that there is no upper bound on message delays in the timeless asynchronous model, a consensus instance can experience runs where it is impossible to progress and thus never terminates. To circumvent this impossibility result, practical consensus algorithms need to provide liveness guarantees in addition to the aforementioned safety properties, as is further discussed in Section 2.3.

2.2 Reconfiguration

A distributed system is, as the name implies, a system consisting of a set $\mathcal{P}$ of processes where all processes execute the same deterministic service. By replicating a service to such a set, one increases the system's initial fault-tolerance. A common scheme to distribute is an atomic storage building on a *read-write quorum system* as shown in [33] and defined in Definition 2.1.

Definition 2.1. A *read-write quorum system* on a finite set of processes $\mathcal{P}$ contains two collections $\mathcal{RQ}$ and $\mathcal{WQ}$ of non-empty subsets of $\mathcal{P}$ $(\forall Q \in \mathcal{RQ} \cup \mathcal{WQ} \colon Q \subset \mathcal{P} \wedge Q \neq \emptyset)$, such that for any two elements $R \in \mathcal{RQ}$ and $W \in \mathcal{WQ}$, $R \cap W \neq \emptyset$ holds. The elements of $\mathcal{RQ}$ are called *read-quorums*, while the elements of $\mathcal{WQ}$ are called *write-quorums*.

Such a system, in its simplest form, consists of two operations: read and write. Each read operation is performed on the processes in a $\mathcal{RQ}$ quorum, and subsequent read operations can either use the same quorum or another quorum found in $\mathcal{RQ}$. Similarly, during write operations a value is written to the processes of a quorum in $\mathcal{WQ}$. The only requirement is that every possible quorum in $\mathcal{RQ}$ intersects every possible quorum in $\mathcal{WQ}$ with at least one process. However, the processes of $\mathcal{P}$ are allowed to fail and processes have a tendency to eventually fail. Failed processes are no longer reachable and can therefore not be part of any quorums. Consequently, if more processes than the system can tolerate fail, the distributed system enters a deadlocked state and can progress no further. In order to maintain the system's fault-tolerance in the face of failures, the set $\mathcal{P}$ constituting the active processes partaking in system operations must be mutable, i.e., allow us to replace failed processes.

In general terms, a system must allow transposition from a current set $\mathcal{P}_c$, initially equal to $\mathcal{P}_i$, to a new set $\mathcal{P}_n$ where $\mathcal{P}_c \cap \mathcal{P}_n = \emptyset \vee \mathcal{P}_c \cap \mathcal{P}_n \neq \emptyset$. Recall that all operations are performed on processes in quorums. Hence, newly added processes, $p \in \mathcal{P}_n \setminus (\mathcal{P}_c \cap \mathcal{P}_n)$, can not contribute to system operations until the quorum system is updated. As a result, a *configuration*, as introduced by [22] and defined in Definition 2.2, consists of both the set $\mathcal{P}$ of processes and its corresponding quorum system.

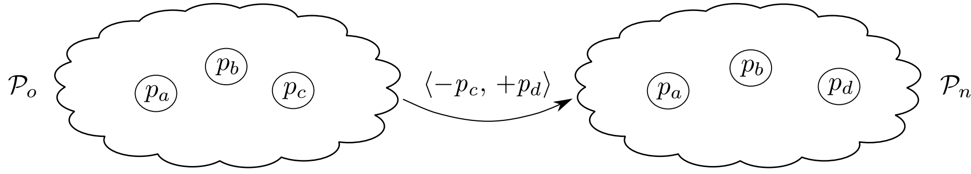


Figure 2.1: A reconfiguration is the procedure of moving to a new configuration. In this case, the reconfiguration consists of removing p_c and adding p_d (i.e., the process p_c is replaced with p_d). For simplicity, changes to the configuration's quorum system is not shown.

Definition 2.2. A *configuration* c is a triple $\langle \mathcal{P}_c, \mathcal{RQ}_c, \mathcal{WQ}_c \rangle$ consisting of a finite set of processes $\mathcal{P}_c$ and a read-write quorum system on $\mathcal{P}_c$, as defined in Definition 2.1.

The procedure of moving to a new configuration, e.g., from $\langle \mathcal{P}_o, \mathcal{RQ}_o, \mathcal{WQ}_o \rangle$ to $\langle \mathcal{P}_n, \mathcal{RQ}_n, \mathcal{WQ}_n \rangle$, is often referred to as a *view change* [2, 3], *group membership change* [12], or a *reconfiguration* [38, 39, 55]. Hereafter, we only refer to such operations as *reconfigurations.* Moreover, we use c to denote a configuration, implicitly denoting a quorum system.

A reconfiguration is simply an operation that mutates the system's current configuration, as illustrated in Figure 2.1. The reconfiguration abstraction can apply a broad range of mutations to the service [33]. It may alter the set $\mathcal{P}$ by removing, adding or replacing processes with subsequent changes to the quorum system. Additionally, it is possible to only replace the quorum system without modifying $\mathcal{P}$ (e.g., optimize the service towards a specific operation).

Reconfigurations can be achieved in one of two ways. The first option is to perform the reconfiguration manually. The service is taken offline, becoming unavailable, and manually redeployed on the new set of machines as specified by the new configuration. The manual approach is far from ideal, manual reconfigurations can be tedious to perform and require human intervention. Furthermore, it disturbs system operations and the system's state is either lost or must be transferred manually to the new machines.

The second, more preferable, approach is to automate the reconfiguration procedure. By relying on sophisticated reconfiguration algorithms, we can remove the human component from the equation and create an autonomous system that

attempts to relieve all of the pain points experienced in the manual procedure. Generally, reconfiguration algorithms aim to have a minimal footprint and to not interfere with normal system operation. Nonetheless, they are actively monitoring the system's health, ready to rapidly react to any unexpected failures. Advanced logic facilitates autonomous state transfer, ensuring consistency among configurations. Preserving the system's availability as it remains online during reconfigurations. However, reconfiguration algorithms are often very advanced and sophisticated, making them hard to grasp and their implementation nontrivial. Incorrect implementations can have severe consequences, including but not limited to availability loss, data inconsistencies and interruption of system operations. As a result, human intervention may be required after all.

Definition 2.3. A *reconfiguration algorithm* is a distributed algorithm with primary focus on reconfiguration logic and provides a modular design resulting in application logic to be loosely coupled from said reconfiguration logic.

Definition 2.4. An *algorithm with reconfiguration capabilities* is a distributed algorithm with the necessary functionality to reconfigure itself. As opposed to *reconfiguration algorithms*, defined in Definition 2.3, this group of algorithm's reconfiguration logic is highly specialized for each use case. Consequently, the reconfiguration logic is tightly coupled, and often intertwined, with application logic. Making it infeasible to adopt the reconfiguration paradigm in another context.

It is important to distinguish between *reconfiguration algorithms* and *algorithms with reconfiguration capabilities*, as defined in Definitions 2.3 and 2.4, respectively. The latter case being defined as a group of algorithms not explicitly aimed at solving reconfiguration, but rather already have the necessities to support such functionality, e.g., Raft [51] and Paxos variants like EPaxos [49]. Commonly, this group of algorithms only vaguely describe their reconfiguration schemes and does not necessarily require nor provide all of the functionality found in reconfiguration algorithms. For example, values chosen for different slots in a consensus instance can be viewed as independent of each other, thus enabling such algorithms to perform reconfigurations without having to transfer the system's state.

As our main focus lies on reconfiguration algorithms, we will not examine nor discuss the reconfiguration techniques of algorithms with reconfiguration capabilities any further. In Chapter 4, we further expand on the topic of reconfiguration algorithms, as defined in Definition 2.3. Here, we closely examine common techniques and classify state-of-the-art reconfiguration algorithms into separate bins based on the different techniques the algorithms employ to achieve reconfiguration.

2.3 The Paxos Protocol

Paxos is an algorithm satisfying the safety requirements of consensus, as specified in Section 2.1. The original paper [39] received a lot of critique for being extremely opaque and hard to understand, motivating the author to publish a new simplified explanation [38]. In addition, there have been multiple attempts to explain the algorithm in simpler terms [41, 42, 47]. The Paxos algorithm has been the topic of extensive research since its introduction, resulting in a wide variety of adaptions such as Disk Paxos [20], FastPaxos [36], Egalitarian Paxos [49], and Vertical Paxos [40].

Paxos is designed for, and proven to be correct in, the asynchronous system model [19]. However, as briefly discussed in Section 2.1, it is impossible to solve consensus even if only one process fails by crashing. To solve this, Paxos utilizes [38] an external leader election module Ω from the timed asynchronous model [15] or the eventually synchronous model [10] to provide liveness guarantees (i.e., eventually progress is made).

In [38], the algorithm is explained through the interaction of three different agent roles. The agent roles can in theory be different processes, but are typically combined in a single process. The agent roles are as follows:

proposer proposes a value to the consensus instance.

acceptor accepts a proposal. If a value is accepted by a large enough set of acceptors, it is said to be chosen.

learner learns a chosen value.

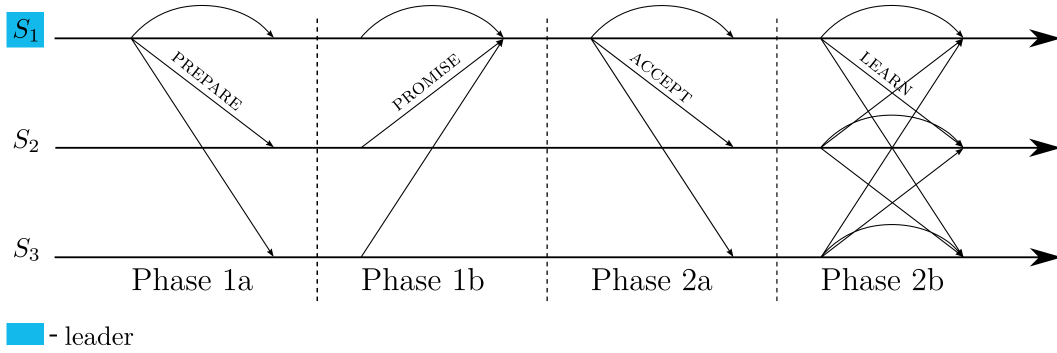


Figure 2.2: An ideal run of the Paxos protocol. The leader, S_1, sends a PREPARE message with a new unique round number in Phase 1a. Upon receiving a quorum of PROMISEs in Phase 1b, S_1 is informed of any previous votes and acknowledged as the global leader for this round. Thus, S_1 can proceed with sending an ACCEPT message containing the safe value in Phase 2a. When a server receives the ACCEPT message, it broadcasts a LEARN to the servers informing them of its decision (Phase 2b). Finally, after a server receives a LEARN from a majority of the servers, the value is said to be chosen.

The algorithm allows a set $\mathcal{P}$ of processes to decide on exactly one value, i.e., exactly one value is chosen. Assume that each process $p \in \mathcal{P}$ contains the logic of all three agent roles. For a value to be chosen, it needs to be proposed by a leader. Due to the system model of Ω, it is possible for several processes to simultaneously believe themselves to be the leader and attempt to propose a value. To ensure consistency in the presence of concurrent leaders, Paxos advances in rounds and requires processes to become the global leader of a round before they are allowed to propose values. Note that if not appropriate care is taken, this scheme can lead to race conditions between multiple concurrent leaders. Additionally, to ensure that only a single proposal is chosen, Paxos requires the proposal to be accepted by a majority of $\mathcal{P}$. We define such a majority as a majority quorum and write $\mathcal{Q} = \mathcal{RQ} = \mathcal{WQ} = \{Q \subset \mathcal{P} \colon |Q| \geq \left\lfloor \frac{|\mathcal{P}|}{2} \right\rfloor + 1\}$. This holds as any two elements of $\mathcal{Q}$ intersects in at least one process, i.e., $(\forall Q_1, Q_2 \in \mathcal{Q} \colon Q_1 \cap Q_2 \neq \emptyset)$, thus it becomes evident that the set of all majority quorums $\mathcal{Q}$ is in fact a valid read-write quorum system on $\mathcal{P}$ in compliance with Definition 2.1. Moreover, we see that such a system tolerates up to f failures for $2f + 1$ processes since a majority of $\mathcal{P}$ consists of at least $f + 1$ processes, e.g., $|\mathcal{P}| = 3 \implies |Q_{min}| = 2$ and $f = 1$.

A round in Paxos consists of two phases, which are illustrated in Figure 2.2. The goals of the first phase are for a process p to become the global leader and to determine if any acceptors have previously voted on a value. Let $\mathcal{S}_Q$ be a finite set of all prior votes reported by a quorum of acceptors and let votes be sortable, i.e., for two votes $v, v' \in \mathcal{S}_Q$ we say $v > v'$ if and only if $v.round > v'.round$. If $\mathcal{S}_Q$ is non-empty, $\mathcal{S}_Q \neq \emptyset$, we say that p is bounded by the previous votes and that the only safe value for this round is the value voted for in $max(\mathcal{S}_Q)$. Otherwise, p is not bounded by any previous votes and all values are safe. For the second phase, the goal is to get the safe value accepted by a quorum such that it gets chosen.

We now give a detailed specification of the phases:

Phase 1.

a) A proposer wanting to propose a value to the consensus instance, (1) checks Ω to verify itself as the leader, (2) creates a new unique round number n that is larger than any previously used round numbers, and (3) sends a PREPARE request with n to at least a majority, $\lceil (|\mathcal{P}| + 1)/2 \rceil$, of processes (acceptors).

b) Upon receiving a PREPARE, an acceptor examines n and acts accordingly. If n is larger than any previously seen round numbers, (1) the acceptor promises to not accept any messages containing a lower round number and (2) replies with a PROMISE containing its latest prior vote, if any. Otherwise, the acceptor is free to discard the message without violating safety.

Phase 2.

a) After receiving a quorum of PROMISE messages, the proposer is acknowledged as the global leader for this round and is informed of any prior votes. Hence, it can proceed with (1) determining the safe value v for this round, and (2) send an ACCEPT message containing the round n and the safe value v to at least a majority of $\mathcal{P}$.

b) When an acceptor receives the ACCEPT message with round n, it (1) accepts the proposal unless it has previously responded to a

message with a greater round number. Upon accepting a proposal, the acceptor (2) broadcasts LEARN messages with n and v to every process in $\mathcal{P}$ to inform them of its decision.

Finally, after a process in $\mathcal{P}$ receives a quorum of LEARN messages with round n and value v, the value v is said to be chosen. Henceforth, any future proposers trying to propose a new value v' is bounded by the chosen value, and as a consequence, must propose v. Thus, no other value v', $v' \neq v$, can become chosen for said consensus instance.

Optimized data-centric Paxos variant. The communication pattern, e.g., all-to-all broadcast of LEARN messages in Phase 2b, makes the Paxos protocol adhere to the process-centric model. A model where servers are allowed to initiate communication among themselves. However, as later explained in Chapter 3, the *Gorums* framework [44] is designed with the data-centric model in mind. In short, the data-centric model only allows clients to initiate communication. Servers only reply to client requests and are not allowed to communicate among themselves. Therefore, to comply with the design requirements imposed by Gorums and the data-centric model, we need to adapt the Paxos protocol without violating the safety properties. Both Disk Paxos [20] and Active Disk Paxos [14] have previously examined and adapted Paxos to the data-centric model. While none of those scenarios are directly applicable to our cause, the developed concepts create a good foundation for further work.

The data-centric Paxos adaption is visualized in Figure 2.3. To simplify the reasoning of its correctness, the adaption is made to closely model the original Paxos protocol, as shown in Figure 2.2. The main difference is the communication pattern enforced by the different models. The data-centric model does not allow for servers to initiate communication, and as a consequence, prohibits server-to-server communication as utilized by Paxos in all phases. In order to comply with the data-centric model, we divide the set of processes into clients, $\mathcal{P}_c$, and servers, $\mathcal{P}_s$, and move the notion of leadership to the clients in $\mathcal{P}_c$. Moreover, only clients in $\mathcal{P}_c$ are allowed to initiate communication, whereas the servers in $\mathcal{P}_s$ are allowed to respond to requests from clients in $\mathcal{P}_c$. The scheme relies on the same majority quorum system as Paxos, the difference being the set $\mathcal{Q}$ is now defined on the

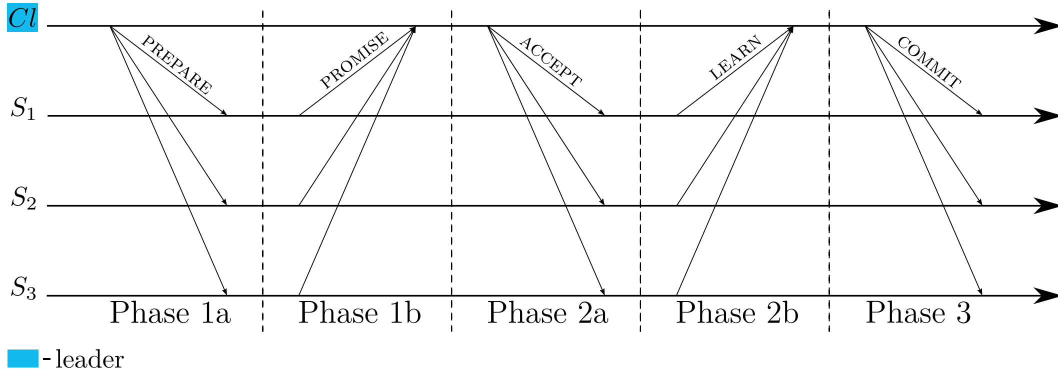


Figure 2.3: The data-centric Paxos adaption closely follows the original Paxos algorithm presented in Figure 2.2. However, to comply with the data-centric model, clients act as leaders and instantiates requests and processes replies. Due to Learn messages not being broadcast to the servers, a third phase is introduced where the client informs the servers that a value has been chosen. Similarly, to deal with concurrent clients trying to propose values, Learn messages now inform the initiator whether the acceptor accepted the value or if the run should be aborted, and Promise messages may return a negative acknowledgement (NAck) if they have previously seen a higher round. If a client receives one or more negative acknowledgements or aborts, it needs to abort the run and try again. The variant is optimized in that it includes the chosen value in NAck messages if the server have previously seen a Commit. Thus, allowing subsequent clients to learn of the chosen value in a single phase, as opposed to completing all three phases.

processes in $\mathcal{P}_s$.

Phase 1 remains almost identical. However, unlike Paxos, we do not rely on an external leader detector Ω. Instead, a client in $\mathcal{P}_c$ can assume itself the leader when it has something to propose. Consequently, in order to avoid race conditions and concurrent clients trying to propose different values, we extend Phase 1b to return a negative acknowledgement NAck when receiving a Prepare message with an outdated round. The NAck message contains the highest round number the server has previously seen and the instance's chosen value, if it is already decided (i.e., has previously seen a Commit message). By including the chosen value in NAck messages after it is decided, we allow subsequent attempts by clients in $\mathcal{P}_c$ to learn of the chosen value in a single round-trip. As opposed to having each client perform all three phases of the algorithm to learn of the chosen value. Clients receiving NAck messages will automatically stop their attempt and back off, either trying

again at a later time or returning the chosen value. Thus, under the assumption that a quorum of processes in $\mathcal{Q}$ are non-faulty and reply in a timely manner, a client will eventually be able to complete the algorithm and get its proposition chosen.

Similarly, we only perform minor modifications to Phase 2. A client p_c continues to be bound by previous votes as in the original Paxos protocol, and starts the phase by sending an ACCEPT message with round n and the safe value to a quorum in $\mathcal{Q}$. However, if a process in said quorum have previously seen a higher round number n', $n' > n$, it will reply with a status of *abort*. Upon receiving such a reply, p_c aborts its current run and reverts back to Phase 1. Due to LEARN messages not being broadcast in an all-to-all fashion among the processes of $\mathcal{P}_s$, a third phase is introduced in which a client informs $\mathcal{P}_s$ that it received a quorum of LEARN messages and that the value is chosen.

From the above specifications, it becomes obvious that the data-centric Paxos adaption adhere to and fulfill the safety requirements of consensus. By utilizing the safe value logic from the original Paxos protocol and a majority quorum system, it follows that only a single value can become chosen and that once a value is chosen, no client will ever be informed otherwise. Additionally, by assuming no non-malicious nor Byzantine faults, the adaption does not allow for non-proposed values to ever become chosen.

3
Gorums

This chapter introduces the *Gorums framework* [43, 44][1]. Gorums is a novel *remote procedure call* (RPC) [7] framework aiming to simplify the process of designing and implementing fault-tolerant distributed systems. Powerful and flexible abstractions allow Gorums to easily invoke RPCs on a set of processes, and to collect and process the replies. First, we introduce the concepts of *Quorum Calls* and *Quorum Functions*. Next, we show the flexibility of the framework by discussing available options and how those make Gorums suitable for a wide variety of distributed algorithms. Finally, insight into the architecture of the framework and a presentation of the underlying technologies are given.

3.1 Quorum Calls

Gorums enables users to easily invoke RPCs on a set of processes [43]. Let Π be the set of available processes. For an RPC invoked on a non-empty subset $\mathcal{P} \subseteq \Pi$, the framework transparently invokes the RPC on every element in $\mathcal{P}$. Returning first after collecting and processing replies from up to a non-empty subset $\mathcal{R} \subseteq \mathcal{P}$ of processes. This is known as a *quorum call* [44], and we say that it returns after receiving a quorum of replies.

Gorums exposes methods for performing quorum calls on any non-empty subset

[1] The Gorums framework has been the subject of another master thesis [52]. Illustrations provided in this chapter are influenced by their work.

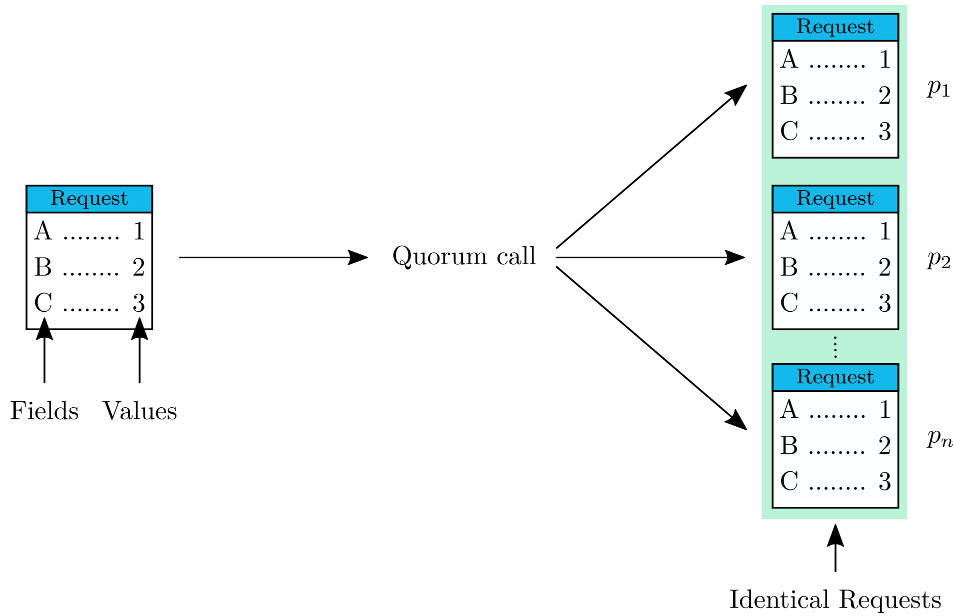


Figure 3.1: The quorum call abstraction enables users to easily invoke RPCs on a set of processes. A quorum call takes a single request as input and transparently invokes an RPC to every process in the set with the same identical request. Replies are collected and transmitted to a quorum function before a single reply is returned to the user.

$\mathcal{P}$, allowing users to easily communicate with any combination of processes in Π. Due to the quorum call abstractions, any non-empty subset $\mathcal{P}$ can be regarded as a single entity from the users point of view. When invoking a quorum call, Gorums transparently invokes an RPC to every process in $\mathcal{P}$, as visualized in Figure 3.1. Their replies are collected and processed, but the quorum call does not deliver the set of replies directly to the user. Instead, the replies are passed to a *quorum function*, where they are combined into a single response returned by the quorum call. Users need to provide quorum functions to accompany every quorum call.

3.2 Quorum Functions

The quorum call abstraction not only collects a quorum of replies, it also processes and combines these replies into a single reply. Thus, the single reply returned by the quorum call contains information about the system as a whole. To illustrate,

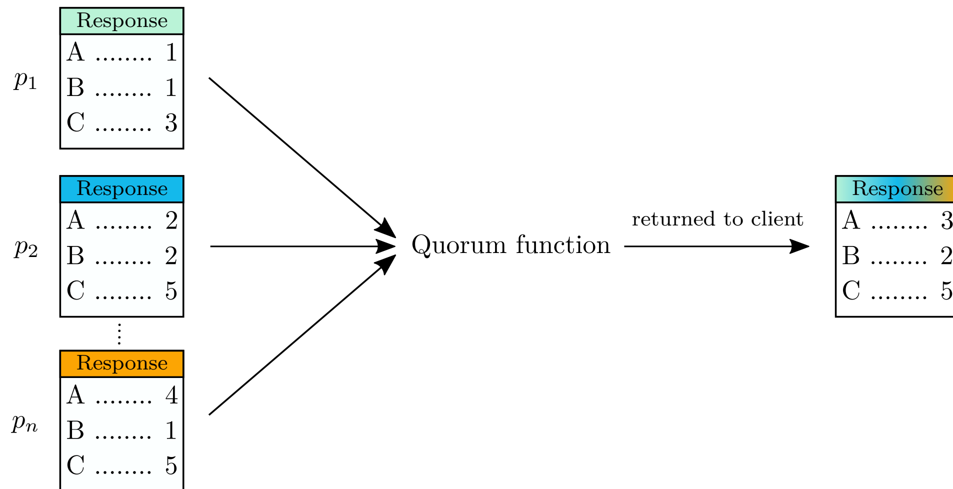


Figure 3.2: A quorum function takes a set of replies and combines them into a single response, which is returned to a calling user. The quorum function is part of a quorum call and have access to a quorum specification to determine whether sufficient information has been received, or if it should await further replies before returning.

a reply from a single process may contain outdated data, whereas the replies from a quorum guarantees that at least one of the replies contains the up-to-date version of the data. To determine if the set of replies $\mathcal{R}$ constitute a quorum and how to combine $\mathcal{R}$ into the single reply, Gorums facilitates user-defined *quorum logic* encapsulated in the quorum function abstraction. Every time a new reply is received, Gorums collects the reply and executes the quorum function on the updated set of replies.

Algorithm 3.1 Default quorum function signature

1: **func** (*qs* QuorumSpec) ExampleQF(*replies* []Response) (Response, Bool)

The default quorum function signature can be found in Algorithm 3.1. The boolean field signals Gorums that a quorum of replies have been received and a meaningful response has been constructed. Once set, the quorum call is said to be complete and Gorums forwards the response from the quorum function to the caller. Alternatively, if the quorum function lacks information to make an informed decision, the boolean field is left negative and Gorums awaits further replies. To prevent endlessly waiting for slow or faulty processes, users can define an upper

time limit for a quorum call. If the quorum function can not make a decision within this limit, Gorums terminates the quorum call and returns an error along with the result from the last execution of the quorum function. Informing the user that a quorum could not be reached. Figure 3.2 visualizes an execution of a quorum function.

As shown in Algorithm 3.1, quorum functions have access to a quorum specification object, QuorumSpec, defining the quorum system in use. Gorums supports a wide range of different quorum systems [60]. For example, for the simplest quorum system, majority quorums, the quorum specification only needs a single parameter denoting the quorum size. Likewise, a quorum specification for a read-write quorum system contains two parameters, denoting the size of their respective quorums (e.g., for a set $\mathcal{P}$ of processes, $\forall Q \in \mathcal{RQ}\colon |Q| \geq 2$ and $\forall Q \in \mathcal{WQ}\colon |Q| \geq |\mathcal{P}| - 1$). Additionally, more advanced quorum systems like *grid quorums*, *latency-efficient quorums*, and *Byzantine quorums* are all achievable by adhering to this scheme, and thus supported by Gorums [43].

Algorithm 3.2 Data-centric Paxos phase 1b quorum function

```
 1: func (qs QuorumSpec) PromiseQF(replies []Promise) (Promise, Bool)
 2:     if replies[len(replies) − 1].NAck then              ▷ latest reply is a NAck?
 3:         return replies[len(replies) − 1], true                   ▷ return NAck
 4:     if len(replies) < qs.QuorumSize then
 5:         return nil, false              ▷ no quorum found, await further replies
 6:     reply := new Promise                          ▷ initialize with empty fields
 7:     for r := range replies do                            ▷ find the safe value
 8:         if r.Vrnd > reply.Vrnd then
 9:             reply = r
10:     return reply, true                                        ▷ quorum found
```

To provide a notion of the power of quorum functions. We provide a working example of the Phase 1 quorum function from the data-centric adaption of Paxos, presented in Section 2.3, in Algorithm 3.2. A client process intending to propose a value to the consensus instance invokes a single quorum call with a new Prepare message, and receives a single Promise message in return. The returned Promise message contains all of the necessary information for the client to proceed in accordance with the algorithm. In reality, Gorums sends an identical Prepare message to every server process partaking in the consensus instance. Each time a

new PROMISE message is received, the quorum function is executed. The quorum function first examines if the newly received reply was a NACK message (L2). If so, it returns the NACK message (L3), completing the quorum call. Otherwise, the quorum function determines if it has received enough replies to constitute a quorum (L4). If a quorum has not been reached, the function returns (L5), signaling Gorums to await further replies. Once a quorum is reached, a single response (L7) is constructed with the safe value (L8-L9), if any, and returned (L10).

3.3 Extensions

To allow for even greater flexibility, the functionality of the quorum call and quorum function abstractions can be modified with a range of options.

Per Node Arguments. Originally, a quorum call on a set $\mathcal{P}$ transmits the same identical request to every process in $\mathcal{P}$. For most algorithms utilizing quorum systems, e.g., atomic storage and consensus, this design paradigm is desirable and adequate. However, there exists use cases where it is preferable to have greater control on the data sent to individual processes. Examples of such cases include erasure coded storage solutions [1, 53] and retransmission of missing information that may result in inconsistencies or prevent a process from participating in the service, e.g., missing entries in a replicated log [51].

To accommodate such scenarios, Gorums provides functionality to execute a user-defined *per server map function* to modify the requests destined to different processes before transmitting them over the wire. The initial request passed to the quorum call is used as a base request by the map function. Unlike quorum functions, whose main responsibilities include combining multiple replies into a single reply, per server map functions utilize the base request to create a new unique request for every process in $\mathcal{P}$. Let $\mathcal{S}$ be the set of unique requests, then the map function can be defined as $f\colon \mathcal{P} \to \mathcal{S}$ and for every process $p \in \mathcal{P}$ the request $f(p)$ is sent to p. Figure 3.3 illustrates how the per server map function works. Notice how different processes receive requests with different values, although only a single request is provided to the quorum call.

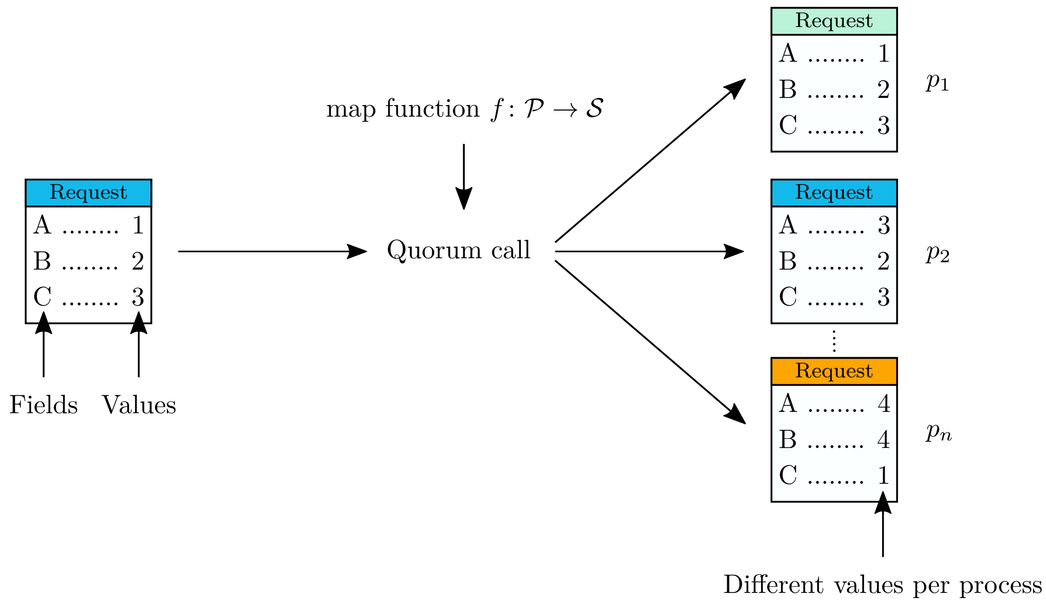


Figure 3.3: A quorum call with the per node arguments option enabled, takes an additional input, a per server map function $f\colon \mathcal{P} \to \mathcal{S}$. The per server map function is user-defined and alters the state of the request sent to different processes according to the user's requirements. Apart from changing the contents of the requests destined for different processes, these quorum calls work similarly to regular quorum calls.

Include Request. There exists several situations where it is desirable to compare and verify parameters in a reply up against a local state, e.g., the round number of received messages in Paxos is compared against a local round number to determine whether the message should be discarded [38]. By default, Gorums does not include the initial request passed to the quorum call in the quorum function, as shown by the function signature in Algorithm 3.1. However, by providing an additional option to the Gorums plugin, the quorum function signature is altered to include the initial request. The modified function signature is shown in Algorithm 3.3.

Algorithm 3.3 Quorum function signature with initial request

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

Custom Return Type. Quorum calls mask the underlying RPCs into a group RPC that can be regarded as a single entity from a user's point of view. A single

argument is provided to the call, and a single response is returned. Any individual errors received during the call are handled internally by the framework, and are only exposed as a single high-level error indicating whether a call was incomplete or timed out. Moreover, quorum calls and quorum functions are limited to the already defined response type of the originating RPC. For most cases, this is adequate. However, if the quorum function is expected to return more information than the information contained in a single reply, e.g., a vector containing a specific field from all replies or any metadata from the processes constituting the quorum, this design scheme is limiting and poses a problem.

A recent work [52] extended the Gorums framework to include a custom response type option. The custom response type is user-defined per quorum call, enabling specific quorum calls to return additional information whereas other quorum calls return the default response type defined by the RPC. By supplying the custom return type option, the signature of the quorum function is modified to that of Algorithm 3.4. Notice how the returned parameter is of type CustomResponse and not Response, as in Algorithms 3.1 and 3.3.

Algorithm 3.4 Quorum function signature with custom return type

1: **func** (*qs* QuorumSpec) ExampleQF(*replies* []Response) (CustomResponse, Bool)

Algorithm 3.5 provides an example quorum function with the custom return type option enabled, and returns additional information about the processes that replied. The quorum function first examines if it has received enough replies to reach a quorum (L2-L3). Once a quorum is reached, a single custom response (L4) is constructed. Then, it inspects each individual reply, storing the identifier of every process partaking in the quorum (L5-L6). Finally, the response is returned (L7) and the quorum call is completed.

Correctables. In latency-efficient quorum systems, it is desirable to return a preliminary result as soon as possible. For example, in EPaxos [49] it is preferable to return as soon as a slow quorum of replies have been collected, and later upgrade this result if further replies arrive and a fast quorum is reached. To provide good support for this type of quorum systems, Gorums provides an implementation

Algorithm 3.5 Example quorum function utilizing the custom return type

```
1: func (qs QuorumSpec) ExampleQF(replies []Response) (CustomResponse, Bool)
2:     if len(replies) < qs.QuorumSize then
3:         return nil, false                  ▷ no quorum found, await further replies
4:     reply := new CustomResponse            ▷ initialize custom type with empty fields
5:     for r := range replies do              ▷ store IDs of nodes participating in the quorum
6:         reply.Ids = reply.Ids ∪ r.Id
7:     return reply, true                     ▷ quorum found, return custom type
```

of the Correctables [31] abstraction. Correctables returns results with incremental consistency guarantees, first returning a fast and possibly inconsistent result, adjusting the result once higher consistency guarantees are available. Hence, Correctables facilitates the creation of latency-efficient applications that are eventually consistent. In Gorums, the option modifies quorum calls to return a correctable, allowing quorum functions to return a stream of incrementally more consistent results.

3.4 Architecture and Implementation Details

Recall that Gorums exposes quorum calls as methods on any non-empty subsets $\mathcal{P}$ on the set of available processes Π, i.e., $\mathcal{P} \subseteq \Pi$. The set of available processes, Π, is registered to a local *manager* object for any given process. The purpose of the manager is to keep track of the available processes, create and maintain network connections, and to monitor said connections, collecting metrics like link-latency and link failures. From the manager, users can create new *configuration* objects.

Gorums allows any combination of processes in Π to be easily grouped together into new configurations. Users simply specify the processes, $\mathcal{P}$, and provide a corresponding quorum specification to create a new configuration c. Quorum calls invoked on c only communicate with the processes in $\mathcal{P}$ and the user-defined quorum specification determines when the call completes. Notice that if the user-defined quorum specification satisfies Definition 2.1, then Gorums' configuration objects are in compliance with Definition 2.2.

As it is the manager's responsibility to maintain and establish network connections, the creation and management of configurations become decoupled from

connection management, i.e., new configurations are not required to perform any connection management and do not experience any network overhead. Thus, it becomes cheap and simple to create and use multiple configurations containing the same set or an overlapping set of processes.

Configuration objects are immutable, i.e., can not be altered nor modified, and are local to a given process. By having configurations as local objects, processes are offered greater flexibility as they can instantiate new configurations on demand and easily communicate with multiple configurations simultaneously. However, the design scheme can be problematic if the system requires a higher level global configuration or if processes need to know whether they are part of any active configuration.

The design of the Gorums framework is heavily inspired by the end-to-end principle [54] and the separation of concerns design principle [17]. A key idea in Gorums is to clearly separate quorum logic from the main control flow of a protocol's operation. [43] showed that quorum calls and quorum functions help in this regard. Normally, different phases in quorum based systems utilize different messages and, possibly, message patterns. Each of these message exchanges can be translated into a quorum call with a corresponding quorum function. By segmenting the protocol logic for different phases into separate quorum calls, and using the system state returned by quorum calls instead of individual replies, [43] found that it greatly simplified the implementation, reasoning, and flow of said systems.

Currently, the Gorums framework is implemented as a library in the Go programming language [59], and its source code is freely available online [45]. As Gorums is written in Go, it follows that all the programming work done for this thesis is also written in Go. The framework works by generating the necessary code to produce an RPC *application programming interface* (API) that allows users to invoke quorum calls on a set of processes. The code generation scheme builds on top of an existing toolchain consisting of gRPC [29] and Protocol Buffers (protobuf) [30]. A short introduction of these technologies are given below. Users specify the desired services and messages in the standard interface description language supported by protobuf in a proto file. Gorums is then supplied as a second-order plugin to the protobuf compiler when processing the proto file, generating both the necessary gRPC and Gorums code. While remaining backwards compatible, the

Gorums generated code wraps and extends the generated gRPC code to expose a new API allowing services to be declared as a group RPC (i.e., allow users to invoke quorum calls on the defined services). As for regular gRPC implementations, developers are still required to and responsible for implementing the server-side RPC handlers. All necessary client-side code is generated and provided by Gorums.

protobuf. protobuf (Protocol Buffers) [30] is a language-neutral, platform-neutral, extensible mechanism for efficiently serializing structured data. The way you want your data to be structured is defined once in a proto file. The protobuf compiler generates a proto file into a special source code, allowing you to easily write and read the structured data to and from a variety of data streams, using a variety of programming languages. Hence, software written in different languages are able to communicate, as they are all able to interpret the structured data.

gRPC. gRPC [29] is an open source high performance RPC framework that enables client and server applications to communicate transparently by transmitting protobuf's structured data. gRPC opens up for defining services on the structured data within a proto file. By enabling the gRPC plugin during compilation, additional code necessary for the RPC services are generated.

4

Reconfiguration Techniques

In this chapter we examine related works and state-of-the-art reconfiguration techniques. We start of by further elaborating on a common keystone in every reconfiguration algorithm: state transfer. First, we present the key idea of state transfer and lay a common foundation to build on top of. Next, we present the three different approaches commonly used to accomplish state transfer in distributed systems. This is followed by a short introduction to availability issues that may arise from incorrect state transfer schemes. Secondly, we divide reconfiguration algorithms into bins, namely, consensus-based and consensus-free reconfiguration techniques. Then, we classify the state-of-the-art reconfiguration algorithms in terms of state transfer and employed reconfiguration techniques. We first present and examine the consensus based schemes, and finally, examine the consensus-free approaches to reconfiguration.

4.1 State Transfer

In terms of reconfiguration, state transfer can be defined as the procedure of transferring a system's state to newly added processes, such that they may start partaking in the system's operations. Recall from Section 2.2 that a reconfiguration is the transposition from an old configuration c_o to a new configuration c_n, i.e., updating both the set of processes actively partaking in the system's operations and its quorum system (reflecting the changes to the set of processes). As pointed

out in Section 2.2, newly added processes usually start of with an empty or blank local state. Therefore, in order to actively partake and be useful to the system, newly added processes may require an updated view of the system's state. The system's state is commonly delivered through the state transfer mechanism to all processes in need of said state. Consequently, the state transfer mechanism becomes an integral part of any reconfiguration operation and, naturally, can be viewed to be tightly linked to the reconfiguration procedure itself. Reconfiguration operations that only alters the quorum system of c, do not necessarily require any state propagation, depending on whether all of the new quorums are already in possession of the up-to-date state.

Processes interacting with newly added processes, before they are representative of the system's global state, may encounter unexpected behaviour such as data inconsistencies, false confirmations and so forth. For example, if a reconfiguration replaces the whole set of processes, communication destined to the new configuration may not read the up-to-date data stored by the previous configuration when read requests arrive concurrently with, or before the state transfer is complete. To counteract and prevent such problems, the reconfiguration algorithm and state transfer mechanism need to incorporate logic to facilitate access in a safe manner, in adherence to all safety guarantees the system is operating within, during this critical period where new processes are brought up-to-date. The state-of-the-art reconfiguration algorithms [2, 3, 13, 22, 35, 58] employ various schemes to cope with this vulnerability. Based upon our analysis of these state-of-the-art algorithms, we have constructed a systematization summarizing their techniques into a total of three generalized methods for performing state transfer, including a brief discussion of the strengths and weaknesses of each method.

The methods are as follows:

1) **Blocking technique**: blocks system operations during state transfer. All regular operations are blocked and clients interacting with the system need to await the installation of the new configuration. The predecessor's processes block access to prevent the system's state from being updated after the state transfer mechanism is invoked, whereas the newly added processes block all regular operations until they have received the updated state and are fully

operational. Thus, circumventing any of the aforementioned problems. The main benefit of this approach is that clients interacting with the system only need to interact with a single configuration at any given point of time. This simplifies the system and reduces the complexity of the clients. Clients do not need to implement complex functionality to aid in or perform either state transfer or reconfiguration tasks. However, these benefits come at the cost of increased latency for blocked operations. The blocking behaviour interrupts system operations and makes the system unavailable for the duration of the reconfiguration procedure.

2) **Wait-free technique**: contact all active configurations (and possible configurations in the consensus-free domain) during state transfer. Unlike the previous approach, the system is always available and does not block any operations during reconfiguration procedures. Instead, the system may have multiple active configurations simultaneously. Hence, a key challenge in this approach is to coordinate the interactions between concurrent operations and reconfigurations, while ensuring availability and maintaining consistency. Clients communicating with the system need to be aware of all active configurations and invoke requests not only to a single configuration, but to all processes in every active configuration. Therefore, it is crucial that clients are able to distinguish and filter replies based on their associated configuration and to verify if a quorum has been formed for every active configuration. It follows that this approach results in clients receiving multiple quorums, and thus, clients need additional complex logic to correctly process, combine, and act on the received data. Moreover, by requiring clients to contact every active configuration, it is apparent that every active configuration adds an additional overhead on system operations.

3) **Concurrent technique**: allows concurrent state transfer operations from multiple processes. When processes encounter new configurations where the state transfer is not yet completed, they initiate the state transfer procedure themselves. As a consequence, the system may experience several concurrent state transfer procedures and needs to act accordingly to maintain safety. However, as for the first approach, processes have the advantage of only

having to interact with a single configuration at any given point in time. After the state transfer is complete, processes do not need to contact the previous configuration and can therefore safely ignore it. Additionally, as processes initiate state transfers themselves whenever they encounter an uninitialized configuration, processes do not need to await other processes completing their tasks. This redundancy does not come without a cost. Disadvantages of the approach include that every process needs to implement the advanced reconfiguration logic, and that the system experiences an even higher overhead than what is the case in the wait-free technique.

The reconfiguration scheme developed as part of this thesis, presented in its entirety in Chapter 5, primarily utilizes the first state transfer approach. The second and third techniques require clients to implement the system's advanced reconfiguration logic and impose additional overhead on system operations. A key strength of systems building on a client server architecture is its language agnostic design. While clients communicating with servers need to conform to an *application programming interface* (API), they do not necessarily have to implement it in the same programming language. Thus, the second and third approach introduce the added challenge of implementing the sophisticated reconfiguration logic in a multitude of languages, which may not be a trivial task. Due to these challenges and the approaches' overhead, we favor the first approach to performing state transfers. However, as we discover in Section 5.1.2, the data-centric model introduces a critical single point of failure in the blocking state transfer technique if appropriate care is not taken. Consequently, we have to borrow some concepts from the third approach to handle concurrent reconfiguration operations and to resolve the single point of failure vulnerability, e.g., when process failures during state transfer operations result in the newly installed configuration being left inactive.

Availability Issues. To highlight a general flaw of all reconfiguration algorithms and state transfer mechanism, we assume a running service utilizing an autonomous reconfiguration algorithm on a set Π of processes. When enough processes fail in rapid succession, i.e., faster than the system is able to reconfigure itself, a window of vulnerability exists [4] until the system stabilizes. In this window, subsequent

failures can result in there no longer being enough active processes to constitute a quorum and, as a result, the system is unable to recover and can progress no further. Thus, it ultimately results in the system becoming unavailable and in need of human intervention. This scenario is a well-known problem that has been addressed in several published works [4, 13, 32, 48].

Successive reconfiguration procedures, either running concurrently or sequentially, may make a system unavailable and unresponsive for a significant amount of time, or in the worst case prevent operations from ever terminating [6]. Additionally, whereas concurrent consensus-based reconfiguration procedures decide on a single successor configuration through consensus, consensus-free schemes have no such limitation and may therefore end up with an unexpected and possibly undesired final configuration [33]. For example, let us assume two processes invoke a reconfiguration simultaneously, adding the processes p_a and p_b, respectively. For the consensus-based solutions, consensus either decides on the new configuration containing p_a or p_b, but not both. If it is desirable to add both, we need to invoke the reconfiguration again with the process that did not make it in the initial reconfiguration. On the other hand, most consensus-free reconfiguration schemes would merge the concurrent proposals in some fashion and end up with a final successor configuration containing both processes. This final configuration can be undesirable and unfavourable, e.g., a majority quorum system running on 3 processes supports a single failure and have a quorum size of 2. The same system running on 4 processes continues to only support a single failure, but have a quorum size of 3. Therefore, additional latency overhead is to be expected. To our knowledge, SmartMerge [35] is the only consensus-free reconfiguration algorithm to address this issue. SmartMerge supports a configuration policy preventing unacceptable configurations from ever being installed.

4.2 Consensus-based Approaches

In general terms, the consensus-based reconfiguration approach consists of two distinctive steps: (*i*) decide on a new successor configuration through consensus, and (*ii*) install the new configuration by utilizing one of the three state transfer mechanisms listed above. Remember that every system starts off with an initial

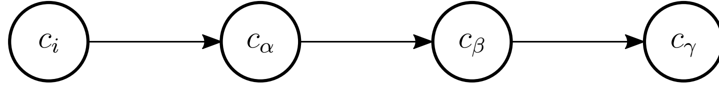


Figure 4.1: Configuration changes throughout a system's lifespan can be modeled by a directed acyclic graph (DAG), showing the successor relationships between the installed configurations. Circles represent configurations and arrows symbolize the successor relationship. Consensus-based reconfiguration approaches use consensus to choose successors for every configuration. As a result, configurations can have at most one successor and there exist a total order between configurations.

configuration and that new configurations are created and activated by reconfiguration operations. We can envision this structure and the set of configurations C as a directed acyclic graph (DAG), as shown in Figure 4.1. Every known configuration, $c \in C$ is a vertex in the graph and the edges symbolize the successor relation between different configurations. Due to the nature of consensus, a consensus instance decides on exactly one value (total ordering property) and, as such, every configuration has at most one successor. Thus, the system's lifespan can be modeled by the DAG as we start off in the initial configuration and traverse the graph as new configurations become available and installed.

Rambo (Reconfigurable Atomic Memory for Basic Objects) [22] presents a consensus-based reconfiguration approach. Admittedly, the presented scheme is not a reconfiguration algorithm per our definition (Definition 2.3), as algorithm operations assist in the reconfiguration procedure. However, trivial modifications to the scheme's garbage collection (state transfer) technique can decouple the reconfiguration logic and make the resulting protocol adhere to our definition. Combining this with the fact that Rambo was the first algorithm to address reconfiguration for atomic storage, and has, as a consequence, been the topic of extensive research [13, 21, 34], we choose to examine the reconfiguration paradigm proposed by Rambo.

Rambo [22] utilizes the second state transfer mechanism listed in Section 4.1, i.e., uses multiple active configurations simultaneously. A reconfiguration automata receives, processes and creates new successor configurations by proposing new configurations to a consensus instance. Due to the state transfer mechanism in use, newly created configurations can be activated instantly without requiring an up-to-

date state. It is the regular algorithm operations that are responsible for garbage collecting old and outdated configurations. In RAMBO, the term garbage collection is equivalent to what we refer to as state transfer. To retain availability during asynchrony, state transfer is invoked lazily, i.e., performed in parallel to regular system operations in the background. The state transfer procedure retrieves the state from old configurations and propagates the information forward to new processes. Once state transfer is complete, old configurations are removed from the set of active configurations (garbage collected), reducing the number of configurations regular operations need to contact.

In [13], Chockler et al presents RDS (Reconfigurable Distributed Storage), an optimized consensus-based reconfiguration protocol advancing on the reconfiguration principles introduced by RAMBO [22]. Accordingly, RDS employs the second state transfer technique, supporting multiple active configurations simultaneously. However, unlike in RAMBO, the state transfer mechanism is tightly coupled to the agreement and installation of new configurations. Due to the lazy approach used by RAMBO, unfavourable conditions may introduce new configurations faster than the garbage collection is able to remove outdated configurations [13, 21, 34]. Resulting in the number of active configurations growing without an upper bound. RDS improves on this by integrating the state transfer mechanism in the reconfiguration procedure. As part of the installation of a new configuration, the processes in the old configuration directly forward their state to the new configuration in an all-to-all fashion. When the state is transferred, the reconfiguration is said to be complete and new algorithm operations are no longer required to communicate with the replaced configuration. Operations that are concurrent to the reconfiguration, i.e., start before the reconfiguration is complete, are still required to contact both the outdated and new configuration and merge the received states.

Compared to the consensus-based reconfiguration scheme devised for this thesis, we find similarities to the techniques used in both RAMBO [22] and RDS [13]. The biggest difference is in terms of state transfer. Both algorithms make use of the second approach to allow for continued availability during reconfiguration, whereas our solution utilizes the first phase and blocks system operations until the procedure completes. Additionally, reconfiguration procedures in our algorithm closely follow the design decision of RDS, combining state transfer with the creation

and installation of new configurations into a single entity. However, it should be noted that RDS uses a process-centric approach, whereas we remain true to the design principles of Gorums and propose a data-centric approach to performing state transfers.

4.3 Consensus-free Approaches

Customarily, reconfiguration schemes rely on the properties of consensus to find and create successor configurations. However, after [2] presented the first consensus-free reconfiguration scheme, a noticeable shift in research culminated in the development of numerous new consensus-free reconfiguration protocols [3, 33, 35, 58]. The consensus-free reconfiguration schemes introduce reconfiguration to services running in the timeless asynchronous domain, a domain where consensus is known to be impossible to solve [19]. It follows that the consensus-free schemes does not rely on consensus to reach agreement on successors. Instead, consensus-free schemes allow configurations to have multiple successors and impose a deterministic convergence property on reconfigurations. The convergence property enforces concurrent proposals to eventually converge in a deterministic manner into a single configuration. Figure 4.2 depicts a generalized consensus-free reconfiguration. Assume multiple processes invoke reconfigurations concurrently, proposing changes to the installed configuration c_i. The convergence property of consensus-free schemes ensure that reconfiguration operations originating from c_i, eventually converge at the same successor configuration. Hence, after continuously traversing the graph and keeping track of concurrent proposals, the reconfiguration operations eventually converge at a final configuration c_{γ_1} and return.

SPSN (Speculating Snapshot) [58] presents a new consensus-free reconfiguration scheme further advancing on the foundation provided by [2]. Similar to [2], SPSN resorts to the third method of state transfer (distinctive processes can initiate state transfer concurrently). The scheme imposes an intersection property on intermediate configurations and merges concurrent proposals. A reconfiguration operation proposes (speculates) its changes to all active and known configurations. Upon the receipt of a proposal, processes return a set containing the active configurations they know of. The reconfiguration operation continues processing such sets

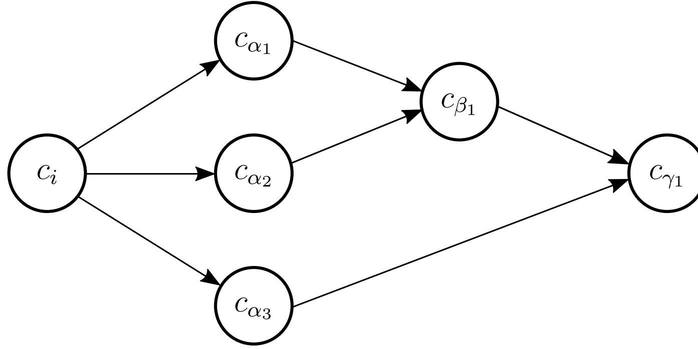


Figure 4.2: Configuration DAG for a consensus-free reconfiguration. Circles represent configurations and arrows symbolize the successor relationship. Consensus-free reconfiguration techniques impose a deterministic convergence property on successors, such that active reconfiguration operations eventually converge on a single vertex, e.g., achieved by combining all concurrent proposals. Note that some consensus-free approaches [3, 35, 58] impose an additional ordering property on configurations, making the configuration DAG one-dimensional (e.g., similar to Figure 4.1 but vertices may have multiple successors.).

until all intermediate configurations have received the proposal. Thus, eventually every intermediate configuration intersects and merges into a single configuration reflecting all concurrent proposals.

FreeStore [3] is another consensus-free reconfiguration scheme continuing on the principles first introduced in [2]. State transfer operations in FreeStore follow the first method listed in Section 4.1, disabling access to the state during state transfer operation. Additionally, the state transfer mechanism closely follows the technique used by Rds [13], where processes directly forward their state to new configurations in a process-centric manner. Configurations in FreeStore are viewed as a sequence, and reconfigurations propose new configurations to the sequence in an incremental fashion, i.e., new configurations are a superset to all previous configurations. Reconfiguration operations continue to update their proposals until it is accepted by a majority quorum. Conflicting proposals are merged together, such that eventually a quorum converges to a sequence containing all proposed configurations (or a combination thereof).

SmartMerge [35] provides another consensus-free reconfiguration technique. By enabling continued availability during state transfers, the scheme conforms to

the second state transfer method listed in Section 4.1. Similarly to FreeStore, the technique imposes an ordering property on configurations, enabling the set of configurations to have a maximum element. Reconfiguration operations propose changes to an auxiliary lattice agreement service. The returned proposal is added to the set of known configurations. The reconfiguration operation chooses the maximum configuration from the set, and writes it to all intermediate configurations. Concurrent proposals and intermediate configurations are merged in accordance to a provided policy. The policy prevents the merging procedure from producing undesired and unacceptable configurations, e.g., configurations with fewer processes than anticipated.

We find numerous similarities between the aforementioned techniques and the consensus-free reconfiguration technique devised as part of this thesis. All techniques incorporate the task of state transfer into the reconfiguration procedure itself and continue on the concepts introduced in [2]. As in [3], we adhere to the first method of performing state transfers, disabling regular operations until the procedure is complete. Further, we employ a timestamping technique comparable to the ordering properties found in [3, 35], enabling us to impose a temporal ordering to the set of configurations. The biggest difference between our approach and the aforementioned schemes is that we do not merge concurrent proposals. Instead, we always reconfigure to the configuration with the highest timestamp. Thus, we ensure that every configuration installed and activated by the system is a configuration actually proposed in its entirety by some process and not an undesired and unacceptable byproduct of merging concurrent proposals, as may be the case in [2, 3, 58]. Moreover, [3, 35] are designed after the process-centric model, whereas similarly to [57, 58], our approach abides by the design principles of the data-centric model.

5 Design and Implementation

In this chapter we present our generalized out-of-the-box reconfiguration scheme. First, we take a look at the design of the devised reconfiguration scheme and further elaborate on the two data-centric reconfigration techniques included in the scheme. Then, we show how our reconfiguration scheme is realized and integrated to function with the Gorums framework [44]. Finally, we present an alternative implementation of the reconfiguration scheme, modifying the behaviour of some modules to take greater advantage of the power provided by Gorums' toolchain.

5.1 Design

The key task of this thesis is to examine and extend the Gorums framework [43, 44] with easy to use, available, and adaptable reconfiguration capabilities. In the following sections, we present and explain the reconfiguration scheme and the two data-centric reconfiguration techniques we have integrated into the framework, reason about their correctness using examples, and highlight design choices made to improve the flexibility of the component.

5.1.1 Reconfiguration Scheme

The reconfiguration scheme needs to be compatible with, and is designed for integration with the Gorums framework [44]. Therefore, we choose to remain true to the design principles of the framework and propose a data-centric approach

to reconfiguration. In the data-centric model, processes are separated into two distinctive roles: clients and servers. A server is a process providing access to an arbitrary service over a communication medium. Servers are only allowed to process incoming requests and may not initiate communication themselves, preventing data-centric algorithms from exhibiting the all-to-all message patterns commonly found in its process-centric counterpart. Clients interact with servers, but not among themselves, utilizing the service provided by servers on users' behalf.

Additionally, to retain the flexibility of the Gorums framework [44] in terms of supported quorum systems, the proposed reconfiguration scheme generalizes on reconfiguration concepts to produce a scheme capable of providing reconfiguration capabilities to a wide range of differing quorum systems. In Chapter 6, we provide working examples for majority quorum and read-write quorum systems, and in Section 5.2.2, we argue for the support of more advanced quorum systems like grid quorums and latency-efficient quorums.

As we aim to provide a generalized and highly flexible reconfiguration scheme, we decouple reconfiguration operations from any protocol operations, complying with our definition of a reconfiguration algorithm in Definition 2.3. Due to this decoupling, the service extended with reconfiguration capabilities is irrelevant and not addressed any further for the remainder of this chapter. We only require the service to be quorum based to fit with the quorum call abstraction of Gorums [44].

We assume a set Π of available servers providing an arbitrary service and a set $\mathcal{C}$ of clients, both sets may be infinite. For the sake of simplicity, we assume that the clients in $\mathcal{C}$ are trustworthy. Subsets of Π are grouped into configurations and subsets implicitly denote the quorum system in use. In addition to its members and quorum system, we assign a unique epoch $e \in \mathbb{N}$ to every configuration. Epochs act similarly to the round numbers in Paxos [38] and are used to uniquely identify each configuration. Further, we use Φ as an abstract entity that is representative of the service's global state, capturing the progress of the service. The details of the global state is highly dependant upon the arbitrary service, and as such, we generalize on the concept of a state and leave it up to the developer to provide the specifics. At any point in time there exists at least one active configuration $\langle c.\mathcal{P} \subseteq \Pi, e \rangle$. Initially, a single active configuration c_i exists, which is known to all clients in $\mathcal{C}$. The configurations in the service's configuration DAG is captured by

the set C of configurations, which initially has the single element c_i. We refer to the set C and the service's configuration DAG interchangeably.

Clients provide a local proxy configuration accessible to users. Changes to the configuration DAG are autonomously and transparently reflected in the local proxy configuration. Thus, users of the system do not need to be aware of the complexity of reconfiguration and always have a single access point they can relate to. Clients interact with servers through the local proxy configuration, and may additionally invoke a reconfiguration operation to change the active configuration running the service. The reconfiguration operation is explained in greater detail in Sections 5.1.2 and 5.1.3.

The reconfiguration scheme works by transparently encapsulating client invocations with additional configuration data Ψ. Configuration data contains metadata representative of the configuration being invoked. For a client invocation on configuration c_i, the encapsulated metadata is $\Psi = \langle c_i.\mathcal{P}, c_i.e \rangle$. Servers decapsulate incoming client requests, inspect Ψ and act accordingly. If $\nexists c \in C$ where $c.e > \Psi.e$, the request is forwarded to the service specific handler. Otherwise, the invoking client is informed of any existing successors $\{c \in C \colon c.e > \Psi.e\}$.

When a client $cl \in \mathcal{C}$ is notified of successor configurations, it executes a local reconfiguration. Local reconfigurations only alter the local proxy configuration at a single client, and do not create successor configurations in the configuration DAG. Local reconfigurations should not be confused with global reconfigurations that are issued by reconfiguring clients to either add, remove, or replace processes partaking in the service's operation.

To illustrate, let us assume a client cl invokes a quorum call for a service request on configuration c_i. During the quorum call, at least one server informs cl about a successor c_α, resulting in cl performing a local reconfiguration from c_i to c_α. Upon triggering a local reconfiguration, the local proxy configuration transparently aborts all outstanding service related requests and starts installing the successor configuration c_α. During installation, the local proxy configuration is modified to target servers $c_\alpha.\mathcal{P}$, implicitly updating its quorum system to reflect the new set of servers. Any service related invocations received during installation are buffered along with the aborted requests. After the installation succeeds, the buffer of pending requests is automatically emptied and invoked on the servers

of c_α. The encapsulated metadata accompanying every request reflects the local proxy configuration at the time of invocation, and as a result, is now equal to $\Psi = \langle c_\alpha.\mathcal{P}, c_\alpha.e \rangle$. Hence, any server that informed the client of c_α, will now accept the request, assuming no successors to c_α have become available in the meantime. Therefore, as long as there is a finite amount of global reconfiguration operations, every invoked service related request is eventually invoked on the active configuration and terminates. From a user's perspective, every invoked quorum call eventually returns and the user remains oblivious to any underlying configuration changes, assuming that fewer than f faults occur during the invocation.

The reconfiguration scheme incorporates state transfer as a vital part of the reconfiguration operation. As a result, the reconfiguration operation consists of first (i) establishing a successor configuration and then (ii) performing state transfer, bringing the successor up-to-date. Servers join and leave the system implicitly through the state transfer mechanism. During the state transfer procedure, the system's state Φ is forwarded to successors. Inactive servers joining the system as part of a reconfiguration, receive a copy of Φ during state transfer and are implicitly activated. Similarly, active servers excluded from the successor configurations will be prompted for a copy of their state during state transfer. After providing a copy of Φ, the excluded servers are implicitly deactivated and receive no further requests from updated clients. Clients with an outdated local proxy configuration may query excluded servers to learn of the successor configuration.

Reconfiguration Interface. The reconfiguration scheme conforms to a $block()$, $release(c_n)$ interface, as illustrated in Figure 5.1. We illustrate the interface using an example. Assume a set of servers provide access to an arbitrary service. The set of servers and its corresponding quorum system is denoted as configuration c_o. Any client interacting with the configuration may invoke a reconfiguration, arbitrarily altering c_o. At the instant a successor configuration c_n is established (i.e., c_n is chosen as the reconfiguration target), the servers of c_o block and disable access to the service. Once the reconfiguration procedure completes, the servers of c_o release c_n, informing any awaiting client of the newly installed configuration. Awaiting clients that are notified of the new configuration continue execution by retransmitting the request to c_n. The reconfiguration procedure is said to be

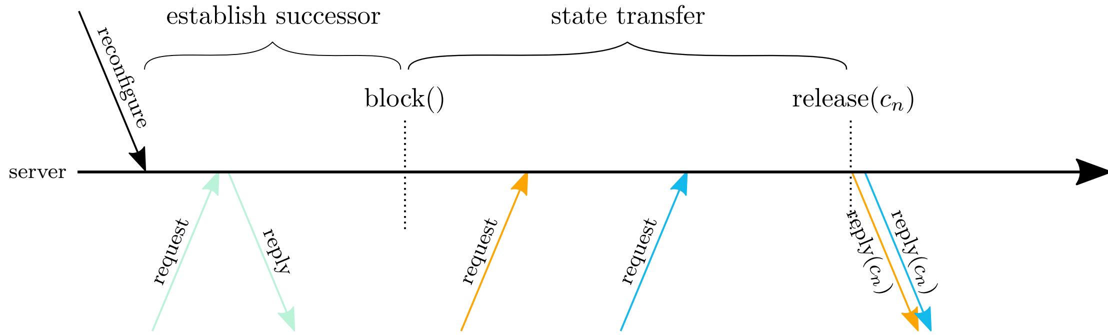


Figure 5.1: The block release interface. Once a successor configuration is established (i.e., there exists a configuration with a higher epoch), previous configurations are blocked, denying access to the service. After the reconfiguration procedure is complete, the successor is released and blocked callers are notified of the newly installed successor.

complete once c_n contains an up-to-date version of the service's state Φ and is ready to process incoming requests. Keep in mind that servers are implicitly activated by receiving a copy of Φ during the state transfer procedure. Hence, when the reconfiguration completes and the servers release c_n, every member of c_n is already activated and ready to process incoming requests.

It is worth pointing out that a reconfiguration operation consists of two steps, (i) agree on a successor configuration, and (ii) bringing the successor configuration up-to-date. The block release interface only limits access to the service for the duration of the second step, preventing clients from modifying the system's state during state transfers.

Reconfiguration API. The reconfiguration scheme extends the functionality of clients to include a new reconfiguration API, enabling clients to propose changes to the active configuration running the arbitrary user-defined service. The API provides functionality to perform single-server membership changes, either adding, removing, or replacing servers from the local proxy configuration.

Although the generalized reconfiguration scheme and both reconfiguration techniques (presented in Sections 5.1.2 and 5.1.3) provide all of the necessary functionality to support arbitrary configuration changes, we decide to restrict the capabilities

of the framework to only include single-server membership changes. There are several reasons justifying our restriction. The single-server membership approach provides the necessary functionality to react to and replace faulty processes, which is the necessary functionality for ensuring continued availability of a service and in retaining its fault-tolerance. Additionally, by only allowing a single membership change between a parent configuration and its successors, we are guaranteed that the servers of these configurations intersects in at least one server. We rely on this intersection to implicitly inform the servers, which are conforming to the block release interface, to release the successor configuration to any blocked clients and future clients.

To illustrate, assume an unknown set of clients $\mathcal{C}$ are actively interacting with the configuration c_α and that a single client $cl \in \mathcal{C}$ invokes a reconfiguration to move from c_α to c_β, where $c_\alpha.\mathcal{P} \cap c_\beta.\mathcal{P} \neq \emptyset$. Once c_β is established, the servers $c_\alpha.\mathcal{P}$ block the incoming requests from $\mathcal{C} \setminus cl$. Eventually, the state transfer completes, implicitly activating the servers of $c_\beta.\mathcal{P}$ and resulting in the servers $c_\alpha.\mathcal{P} \cap c_\beta.\mathcal{P}$ releasing c_β to the blocked clients. Any client that did not interact with c_α during the reconfiguration, will upon the next invocation be informed of c_β from the servers $c_\alpha.\mathcal{P} \cap c_\beta.\mathcal{P}$.

It is easy to realize arbitrary changes through the single-server membership approach by performing sequential reconfigurations, replacing one server at a time. However, it remains trivial to extend the reconfiguration API to include support for single operation arbitrary configuration changes. We only need to extend the reconfiguration techniques to include an additional activation message, signaling the blocked configuration to release the new successor configuration. The message should be sent once the state transfer procedure is complete.

Allowing every client to perform global reconfiguration operations may be problematic and has some security implications that should be carefully considered. For a system model that assumes every client to be trustworthy, it is fine to expose access to the reconfiguration API to every client. However, if there is a possibility of clients having a malicious intent, the access should be restricted to a few trustworthy clients. We envision that in such cases the processes we refer to as clients are not necessarily endpoint clients available to users. Instead, we envision them to be the trustworthy clients with access to the reconfiguration API. Such clients may for

example be intermediate reconfiguring clients as part of a monitoring component that monitors the system's health and invokes reconfigurations in response to unexpected faults. Regardless of the case, it remains up to the developer to restrict access to the reconfiguration API, only exposing the desired functionality to endpoint clients available to users.

Finding New Configurations. The block release interface and the membership intersection property imposed by the reconfiguration API are all designed for helping clients traverse the configuration DAG, informing them of any configuration changes. It is reasonable to assume that for any practical system, reconfigurations are infrequent and far in between. Reconfigurations are usually invoked to replace faulty processes or to scale the system to handle increasing load. Hence, it is not unlikely that a server known to new or inactive clients is a member of several configurations and can help clients traverse the DAG, bringing their local proxy configuration up-to-date.

Nonetheless, the possibility of the active configuration becoming unreachable for inactive or new clients is ever-present. For example, assume an initial configuration c_i where $c_i.\mathcal{P} = \{p_1, p_2, p_3\}$. After a minimum of six reconfiguration operations, every member of c_i may be replaced with new servers. Let c_γ be the resulting configuration of the sixth reconfiguration and let $c_\gamma.\mathcal{P} = \{p_4, p_5, p_6\}$. Thus, every client in $\mathcal{C}$ that were inactive and did not learn of any successors to c_i that have a non-empty intersection with c_γ, will from this point forward be unable to interact with the service as c_γ is unreachable to them. This is an issue common to all distributed systems dealing with dynamic services [57, 58]. As the focus of this thesis is to introduce reconfiguration capabilities to arbitrary services, this problem is beyond its scope and we do not explicitly provide a solution. However, it could be solved by employing some sort of distributed directory service, or oracle, that stores information about configurations. An example of such a distributed global directory service is *domain name system* (DNS). Clients query the directory or oracle for information, learning of new configurations as required. Likewise, reconfiguration operations inform the directory or oracle of the resulting successor configuration before returning.

Implementation Requirements While the implemented service we extend with reconfiguration capabilities is irrelevant, its state Φ is not. Due to the nature of a service's state, being highly dependant on the implemented service, it is impossible to generalize a one solution fits all when it comes to the extraction and installation of Φ. Therefore, to be compatible with the reconfiguration scheme, we require developers to define two additional quorum calls to get and set Φ. We envision the use of distinct options to inform Gorums [44] of the state related quorum calls during compilation. The get state quorum call is required to return a copy of Φ, capturing the relevant information necessary to propagate the service to new servers. Likewise, we require the set state quorum call to accept a batch of messages returned by the get state quorum call, propagating them to the configuration and returning an acknowledgement informing the quorum call that it succeeded. As for regular Gorums implementations, developers need to implement the corresponding quorum functions and the server-side implementations to handle incoming requests.

Moreover, as explained in Section 5.2.2, Gorums requires every configuration object to be accompanied by a quorum specification. To be able to transparently alter the underlying configuration, we need to be able to automatically update the quorum specification to reflect changes in the local proxy configuration. Therefore, we also require developers to provide a quorum specification factory method that is capable of creating new quorum specifications on demand.

5.1.2 Consensus-based Approach

We now present our consensus-based reconfiguration approach. The approach fits into the aforementioned reconfiguration scheme as the client-side reconfiguration operation, also previously referred to as a global reconfiguration.

The consensus-based approach consists of a sequence of instances of consensus $P_1, P_2, P_3, \dots$. Every configuration is accompanied by a unique consensus instance capable of agreeing upon exactly one successor. Each client proposes their changes to instance P_k for the kth configuration c_k. Consensus instance P_k utilizes a majority quorum system on configuration c_{k-1} to agree on the new configuration c_k, which is made available to the reconfiguring clients. Reconfiguring clients use c_k as the reconfiguration target, bringing it up-to-date and implicitly activating it

through the user-specified state transfer quorum calls. As for local reconfigurations, the technique aborts all outstanding service specific requests and buffers any incoming service specific requests for the duration of the reconfiguration operation. Once the successor is successfully installed, any aborted or buffered requests are retransmited to the new configuration.

Algorithm 5.1 Data-centric consensus-based reconfiguration

```
 1: State:
 2: cur                                              ▷ local proxy configuration

 3: reconf(change):
 4:     abort all outstanding requests and buffer incoming requests
 5:     new ← ⊥                                       ▷ reconfiguration target
 6:     S ← {}                                        ▷ set of states, initially empty
 7:     new ← cur.consensus.propose(change, cur.epoch + 1)   ▷ establish successor
 8:     S ← S ∪ cur.getState()                        ▷ collect state Φ
 9:     cur ← new                                     ▷ update proxy
10:     if ∄Q ∈ cur.setState(S) then                  ▷ no quorums to set state on target
11:         return reconf(newTarget)                  ▷ reconfigure according to active policy
12:     retransmit aborted and buffered requests to cur
```

Algorithm 5.1 shows a detailed presentation of our consensus-based reconfiguration technique. Parts of the technique is generalized to streamline common functionality shared between the consensus-based and the consensus-free technique. For example, the consensus-based approach only collects the state Φ from a single configuration and it can therefore be seen as excessive to treat the single Φ as a set. This design allows both techniques to share lots of code, increasing code reusability and reduces the overall code complexity. Additionally, one could argue that the technique could be optimized and made more efficient by aborting reconfiguration operations at the clients that did not get their proposals chosen by consensus. Applying such an optimization reduces the number of redundant messages experienced during concurrent reconfigurations from $(|c_\alpha.\mathcal{P}| + |c_\beta.\mathcal{P}|) \times (n-1)$ to 0, where n is the number of concurrent reconfigurations. However, the optimization has a detrimental effect on the overall liveness of the system, creating a single point of failure that can render the complete system unresponsive and unavailable. Let cl be the reconfiguring client that got its proposition chosen by consensus. Every reconfiguring client except cl aborts their reconfiguration operation as per the optimization. cl proceeds with the reconfiguration and collects the system's state

from the parent configuration. Recall that servers comply with the block release interface and started blocking service specific requests as soon as the successor was established. Thus, the system is unresponsive and blocks incoming requests from every client except cl. If cl was to crash before successfully installing the collected state on the successor, the servers would never be activated and therefore never release the new configuration. Effectively making the system unresponsive and blocking indefinitely. All non-faulty clients will be unable to complete the reconfiguration as their proposals were not chosen by consensus, aborting their reconfiguration attempts altogether. Should a similar scenario occur without the optimization in place, non-faulty clients can simply learn the reconfiguration target by proposing any value to the consensus instance and try to install the target. It would be interesting to investigate whether we could bypass this problem while applying the optimization by adopting a lease-based approach [46]. Unfortunately, due to time restrictions, we leave this investigation as an interesting topic for further work.

To extend the flexibility of the reconfiguration scheme and add support for different quorum systems, the technique includes an autonomous reconfiguration component (L10-L11) capable of executing an autonomous reconfiguration in accordance to a given policy. The set and get state quorum calls are user-defined, and as such, utilize a user-defined quorum specification. Should the set state quorum fail, i.e., no quorum is reached as per user specifications, a successive reconfiguration is launched, executing the active policy to create a new reconfiguration target. Thus, trying to ensure continued availability of the service. For example, to add support for all read-write quorum systems, a reconfiguration policy removing faulty servers is sufficient. If the quorum call is unable to reach consensus, one assumes that unresponsive servers are faulty and omits them from the new target. Note that if half the servers have failed, the consensus-based technique fails as the consensus instance is incapable of reaching consensus. Consequently, the autonomous reconfiguration attempt fails and the system is forced to stop making progress indefinitely.

The devised scheme is capable of supporting a wide range of policies, altering the characteristics of the component. Unless the scheme relies on an auxiliary oracle service (e.g., DNS) to distribute configuration data, policies should not

violate the intersection property we impose on configurations. A policy should be treated as a last resort in trying to alleviate the system and resolve a potentially deadlocked state. Further, it is not given that a policy that is safe for a quorum system does not violate protocol properties in another quorum system. Hence, the development and design of additional policies is an interesting and complex topic for further work.

Consensus Instance. The consensus-based reconfiguration technique is compatible with all consensus algorithms. It is trivial to change the consensus instance, switching algorithm or altering the scheme to rely on an external auxiliary consensus service. In order to abide by the data-centric model and conform to the design choices of Gorums [44], we incorporate our optimized data-centric Paxos variant as the included default consensus instance. Section 2.3 introduces our data-centric Paxos variant. Further, a complete run of the algorithm is visualized in Figure 2.3 and Algorithms 5.2 and 5.3 depicts the Paxos variant in pseudocode format.

The Paxos variant enables clients to propose values to the consensus instance that may be voted for. Eventually, a single proposal is chosen by the instance. Subsequent proposals return the chosen value. Proposals are accepted by the **propose**(*proposal*) method listed in Algorithm 5.2. Each client may propose at most a single value to the consensus instance. If a client attempts to propose a second value, the invocation should await the first proposal and return the chosen value. Clients without any values to propose, simply refrain from invoking the method. The algorithm is persistent, only returning after a chosen value has emerged. To this extent, the algorithm utilizes exponential back-off to prevent race conditions between concurrent proposers, eventually allowing a single client to get a value chosen.

After receiving a proposal, a client starts the first phase of Paxos and attempts to become global leader for a round number n (L7-L17). A client is said to win the round once it receives a confirmation from a majority of the servers. Due to a possibly infinite number of clients, we can not guarantee that a round number used by a specific client is unique. However, by restricting servers to require incremental round numbers and only send a confirmation the first time a particular round number is received, we guarantee that if a client wins the round then no other

Algorithm 5.2 Client-side data-centric paxos

```
 1: State:
 2: cur                                                         ▷ the configuration
 3: chosen                                                      ▷ the chosen value

 4: propose(proposal):
 5:   sval ← (⊥, ⊥)                                             ▷ the safe value
 6:   rnd ← generate a random round number from a finite set of possible values
 7:   repeat
 8:     prm ← cur.propose(rnd)                                  ▷ quorum call to ∀p ∈ cur.P
 9:     if ∃nack ∈ prm do                                       ▷ received a negative acknowledgement
10:       if ∃chsn ∈ nack do                                    ▷ a value is already chosen
11:         cur.commit(chsn, nack.rnd)                          ▷ quorum call to ∀p ∈ cur.P
12:         chosen ← chsn
13:         return chosen
14:       rnd ← nack.rnd
15:       exponential back off
16:     rnd ← rnd + 1
17:   until ∃Q ∈ prm ∧ ∄nack ∈ Q                                ▷ won the round
18:   if (vrnd = ⊥) ∀(vrnd, vval) ∈ prm do
19:     sval ← (proposal, rnd)                                  ▷ freely choose a value
20:   else do
21:     sval ← safe value ∈ prm                                 ▷ bounded by a safe value
22:   lrn ← cur.accept(sval, rnd)                               ▷ quorum call to ∀p ∈ cur.P
23:   if ∄Q ∈ lrn ∨ ∃abort ∈ lrn do                             ▷ did not receive a quorum of lrn
24:     exponential back off and restart phase 1
25:   cur.commit(sval.Value, rnd)                               ▷ quorum call to ∀p ∈ cur.P
26:   chosen ← sval.Value
27:   return chosen
```

client may successfully win the round with an identical or lower round number. If a server know of a higher round number n', $n' > n$, it responds with a negative acknowledgement NACK. A NACK contains the highest round number known to the server and the chosen value, if any. Consequently, a client receiving a NACK either backs off exponentially (L15-L16), increasing the timeout between attempts, or if the chosen value is present, returns the chosen value (L11-L14). Once a client has won the round, it determines the safe value (L18-L21) and proceeds with the second phase of Paxos (L22-L24). If the second phase is successful, the client commits and returns the value (L25-L27). On the other hand, should a server return an ABORT or if the client fails to get a quorum of replies, the client backs off exponentially and restarts the algorithm, reverting back to the first phase of Paxos (L23-L24).

Algorithm 5.3 Server-side data-centric paxos

```
 1: State:
 2: rnd                                              ▷ the round number, initially ⊥
 3: vval                                             ▷ the voted value, initially ⊥
 4: vrnd                                             ▷ last voted round, initially ⊥
 5: chosen                                           ▷ the chosen value, initially ⊥

 6: on ⟨PRP: round⟩ from p                           ▷ received prepare
 7:     if round ≤ rnd ∨ chosen ≠ ⊥ do
 8:         p ← ⟨NACK: rnd, chosen⟩    ▷ outdated, return a negative acknowledgement
 9:     rnd ← round
10:     p ←⟨PRM: rnd, vrnd, vval⟩                    ▷ return a promise

11: on ⟨ACC: val, round⟩ from p                      ▷ received accept
12:     if round < rnd ∨ round = vrnd do
13:         p ← ⟨ABORT⟩       ▷ outdated round or previously voted in this round
14:     vrnd ← round
15:     vval ← val
16:     p ← ⟨LRN: rnd⟩                               ▷ return a learn

17: on ⟨CMT: val⟩ from p                             ▷ received commit
18:     if chosen = ⊥ do
19:         chosen ← val
```

In the data-centric Paxos variant, servers remain passive and only respond to client requests. Consequently, servers do not communicate with one another and do not need to know of each others existence. Due to the data-centric nature, servers need to rely on clients behaving and acting according to the algorithm. Thus, by employing the data-centric Paxos variant, the reconfiguration scheme is incapable of correctly extending byzantine services with reconfiguration capabilities. Being byzantine safe is not in the scope of this thesis.

The server-side functionality of the algorithm is captured in Algorithm 5.3. The algorithm only deviates slightly from single-decree Paxos [38, 39] for the first and second phase. Instead of ignoring outdated requests, it returns a negative acknowledgement NACK (L8) or an abort ABORT (L13), in the first and second phase, respectively. Additionally, the Paxos variant includes functionality to handle the newly introduced third phase (L17-L19). The first time a server receives a COMMIT, the chosen value is set and subsequently included in any future NACKs. Effectively preventing successive or concurrent clients from completing more than the first phase of Paxos to learn the chosen value.

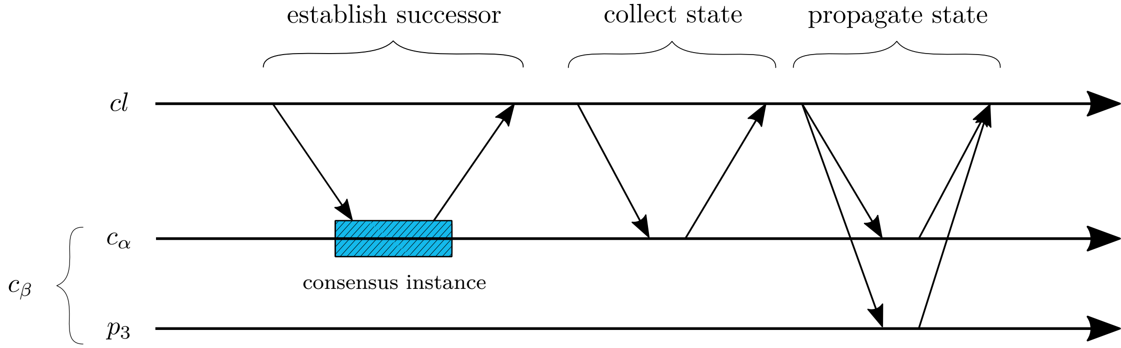


Figure 5.2: The consensus-based reconfiguration technique utilizes a consensus instance to establish the successor configuration. Once a successor is available, the reconfiguration operation collects the system's state and propagates it to the new configuration, bringing it up-to-date and ready to serve incoming requests.

A Single Reconfiguration. We now explain how the consensus-based approach performs a single reconfiguration from configuration c_α to c_β. Let us assume that the reconfiguration adds a new server p_3 to the system running the arbitrary service. The reconfiguration technique is detailed in Algorithm 5.1 and its message pattern is visualized in Figure 5.2.

A global reconfiguration, altering the servers constituting a service, is initiated by a client $cl \in \mathcal{C}$ locally invoking its reconfiguration operation $\mathbf{reconf}(\langle +p_3 \rangle)$. As a precaution, the invoker enters a locked down state (L4) preventing it from initiating any non-reconfiguration related communication for the remainder of the operation. The precaution ensures that every invoked quorum call is always executed on the most up-to-date configuration known to the client. After entering the locked down state, *cl* proceeds by proposing $\langle +p_3 \rangle$ to the current configuration's consensus instance (L7). If no other clients in $\mathcal{C} \setminus cl$ are simultaneously performing or have previously performed a reconfiguration, *cl*'s proposition is uncontested and eventually chosen by consensus. Thus, every client in $\mathcal{C}$ performing a subsequent reconfiguration on c_α will learn of the successor configuration c_β incorporating *cl*'s proposal ($c_\beta.\mathcal{P} = c_\alpha.\mathcal{P} \cup p_3$).

Now, the first phase of the reconfiguration operation is complete and *cl* has established a common reconfiguration target, available to all reconfiguring clients. In this case, *cl* sets the established successor c_β as the reconfiguration target and attempts to activate the configuration by completing the state transfer procedure

(L8-L12). Before being able to bring c_β up-to-date, *cl* needs to collect a snapshot representative of the system's current state at the time of invocation. By invoking the user-provided get state quorum call (L8), *cl* is able to query every server in c_α for a copy of their local state, capturing the necessary information required for representing the service's state at the time of invocation. If the quorum call succeeds, i.e., receives enough replies to satisfy the user-provided quorum requirements, *cl* updates its local proxy (L9) and propagates the information forward to the successor configuration (L10), bringing it up-to-date and ready to process incoming requests. Finally, after successfully completing the state transfer procedure, *cl* unlocks and retransmits any aborted or queued invocations to the new configuration c_β (L12). Due to the set state quorum call implicitly activating the new configuration, we require every server in the new configuration to acknowledge its receipt. In the future, it would be advisable to optimize the set state quorum call to make use of Gorums' per server map function to transmit an empty activation message to the servers in $c_\alpha.\mathcal{P} \cap c_\beta.\mathcal{P}$ as they already contain an up-to-date copy of the service's state.

On the off-chance of the get state quorum call failing, it is reasonable to assume that the number of faulty servers exceed the fault-tolerance of the system, resulting in the service being unavailable indefinitely and in need of human intervention to proceed. Likewise, the assumption holds for the set state quorum call. However, *cl* is already in possession of an up-to-date state capturing the service and can therefore initiate a new reconfiguration according to a given policy (L11) in an attempt to autonomously restore the system's fault-tolerance and allow for continued availability of the service. As quorum calls are user-provided and highly dependant upon the implemented service, no further details of their inner workings are provided in this chapter and it is up to the user to implement it correctly.

Concurrent Reconfigurations. Due to reconfiguration operations being local to clients, they have no way of knowing whether other clients are currently performing the first step of reconfiguration (trying to establish a successor). Consequently, multiple clients may invoke the reconfiguration operation simultaneously. The consensus-based reconfiguration technique utilizes the power of consensus to guarantee that every configuration has exactly one successor. Let c_α be the current

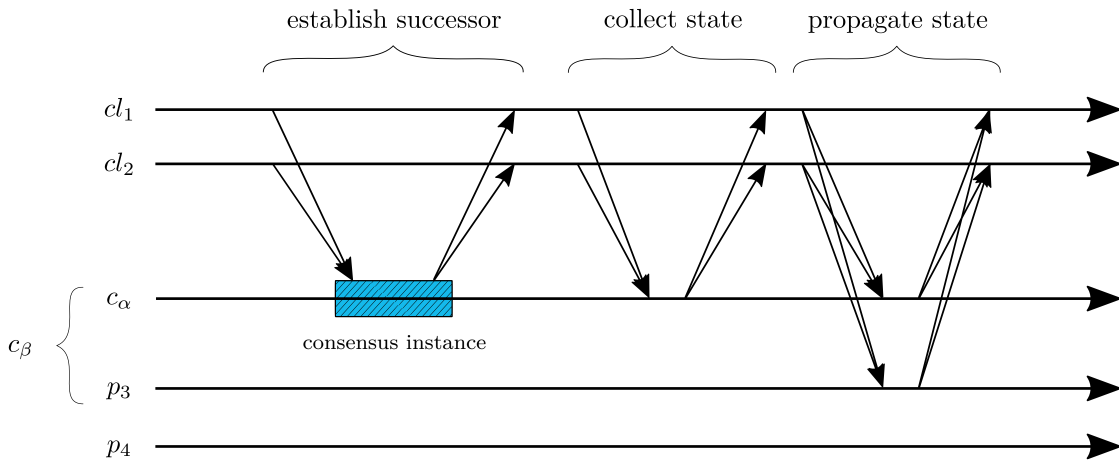


Figure 5.3: In the consensus-based technique, concurrent reconfigurations propose their changes to the consensus instance. Once a successor is available, every reconfiguring client targets the successor and attempts to bring it up-to-date through the state transfer procedure. The server p_4 is not a member of the successor configuration and consequently left out of system operations.

configuration. Concurrent reconfigurations on c_α only increase the set of proposals available to c_α's consensus instance, implicitly increasing the number of possible successors. Eventually, exactly one proposal c_β is chosen and every concurrent and subsequent reconfiguration operation on c_α attempts to install c_β. Figure 5.3 depicts a scenario where two clients concurrently invoke a reconfiguration towards different targets.

The maximum successor restraint imposed by consensus enforces every concurrent reconfiguration to perform a possibly superfluous state transfer procedure. Nonetheless, every concurrent reconfiguring client increases the overall redundancy of the reconfiguration operation, increasing the likelihood of success and circumventing the critical single point of failure discussed previously. To elaborate, even though a reconfiguring client fails after completing L8 but before executing L10, triggering a limbo where both parent and successor configurations are left inactive, a concurrent or subsequent reconfiguration invoked by another client may succeed in propagating the state forward to the successor and thus ensuring continued availability of the service. Additionally, if there is no concurrent reconfiguration operations, clients can detect the blocking behaviour of the servers and deduce the system's deadlocked status, triggering a subsequent reconfiguration to resolve the

scenario.

Finally, it is worth pointing out that reconfiguring clients have no guarantee of getting their proposal chosen as the reconfiguration target. Therefore, it may be necessary for unsuccessful clients to perform consecutive reconfigurations to finally get their proposal included in the active configuration.

5.1.3 Consensus-free Approach

We now present our consensus-free reconfiguration approach. Similar to the consensus-based technique, the consensus-free approach fits into our reconfiguration scheme as the client-side reconfiguration operation (global reconfiguration). The two techniques are interchangeable, enabling users to choose their preferred technique based on service requirements. Further, the consensus-free approach complements our reconfiguration scheme, making the scheme suitable for environments in which consensus is deemed impossible to solve. Neither techniques require any additional user-provided functionality nor introduce any new demands to users in order to function properly.

The consensus-free approach does, as the name implies, not utilize consensus as part of the reconfiguration operation. Without consensus, it is impossible to guarantee that a configuration has at most one successor. Instead, every reconfiguration can introduce new successors that need to be accounted for. Therefore, the devised technique requires a convergence property, ensuring that every concurrent reconfiguration eventually reaches an identical configuration by traversing the configuration DAG. In order to limit the number of successors experienced during concurrent reconfigurations, our devised consensus-free approach draws inspiration from the techniques introduced in [57] and further builds on the concept of timestamped reconfigurations utilized by ARec [33]. By imposing a temporal order on configurations, we can define the convergence property to always target the newest configuration known to the reconfiguration operation. Thus, eventually every reconfiguration operation that continuously traverses the configuration DAG converges to the latest configuration proposed to the system. Introducing temporal order on configurations is achieved by adding stricter constraints to the configuration's epoch. In the consensus-free approach, valid epochs are required

to be unique and incremental, but not necessarily consecutive. For example, if the highest epoch used in the system is n, any epoch in the set $\{n' \colon n' \in \mathbb{N} \land n' > n\}$ conforms to the requirements and is therefore valid.

In the consensus-free approach, clients propose changes to their local proxy configurations. To help reconfiguring clients traverse the reconfiguration DAG, servers receiving configuration proposals inform the proposing client of any relevant successors they have previously seen, if any. When informed of unknown successors, reconfiguring clients inspect the successors to determine whether they are more up-to-date, possibly changing the reconfiguration target, and propose said target to every newly learned successor. This process repeats itself until there are no more successors to contact. Hence, for a finite number of reconfigurations, eventually every reconfiguring client targets the same identical configuration and can bring it up-to-date, implicitly activating it through the user-specified state transfer quorum calls. As for local reconfigurations, the technique aborts all outstanding service specific requests and buffers any incoming service specific requests for the duration of the reconfiguration operation. Once the reconfiguration target is successfully installed, any aborted or buffered requests are retransmitted to the newly installed configuration.

Note that unlike the consensus-free reconfiguration techniques examined in the literature (Section 4.3), our consensus-free technique does not merge successors but rather uses a single distinct proposal as the reconfiguration target. If the reconfiguration operation encounters a successor configuration with a higher epoch, we simply switch the reconfiguration target and proceed accordingly. Hence, we ensure that every installed configuration is actually a configuration proposed by some client, and not an undesired byproduct of some merging behaviour, possibly leaving the system in a reduced state with lower fault-tolerance or increased overhead. Moreover, if a reconfiguring client crashes during the aforementioned critical single point of failure, resulting in the system finding itself in a deadlock, other clients can detect the blocking behaviour of the servers and subsequent deadlock and attempt to resolve it. By triggering a new reconfiguration targeting the currently installed proxy that reuses its corresponding epoch, this reconfiguration operation traverses the configuration DAG and learns of the more up-to-date inactive successor configuration. After learning of the successor, the client can attempt

Algorithm 5.4 Data-centric consensus-free reconfiguration (Client)

```
 1: State:
 2: cur                                                        ▷ local proxy configuration

 3: reconf(change, epoch):
 4:     abort all outstanding requests and buffer incoming requests
 5:     new ← (change, epoch)                                  ▷ reconfiguration target
 6:     maxActive ← 0                         ▷ highest encountered active configuration
 7:     C ← {cur}                                          ▷ set of unknown successors
 8:     T ← {cur}                                     ▷ set of tracking configurations
 9:     S ← {}                                        ▷ set of states, initially empty
10:     while |T| > 0 then
11:         c ← c ∈ T
12:         T ← T \ c
13:         if c.epoch > maxActive ∧ c.active = true do
14:             maxActive ← c.epoch
15:         confs ← c.propose(new, cur)                          ▷ propose target to c
16:         for conf ∈ confs ∧ conf.epoch > cur.epoch do   ▷ unknown successors to cur
17:             if conf.epoch > new.epoch then                    ▷ successor to new
18:                 return reconf(conf.P, conf.epoch)  ▷ change reconfiguration target
19:             C ← C ∪ conf
20:             T ← T ∪ conf
21:     for c ∈ {c ∈ C: c.epoch ≥ maxActive} do
22:         S ← S ∪ c.getState() in parallel                   ▷ collect state Φ for c
23:     cur ← new                                                     ▷ update proxy
24:     if ∄Q ∈ cur.setState(S) then             ▷ no quorums to set state on target
25:         return reconf(newTarget, e > cur.epoch)  ▷ reconfigure according to active policy
26:     retransmit aborted and buffered requests to cur
```

Algorithm 5.5 Data-centric consensus-free reconfiguration (Server)

```
1: State:
2: epoch                                                       ▷ highest known epoch
3: C                                                    ▷ set of known configurations

4: on ⟨PRP: new, old⟩ from p                ▷ received a new configuration proposal
5:     if new.epoch > epoch do
6:         epoch ← new.epoch                   ▷ signals the block release interface
7:     C ← C ∪ new                              ▷ add to set of known configurations
8:     R ← {c.P, c.epoch, c.active: c ∈ C ∧ c.epoch > old.epoch}
9:     p ← R                                  ▷ return the set of relevant successors
```

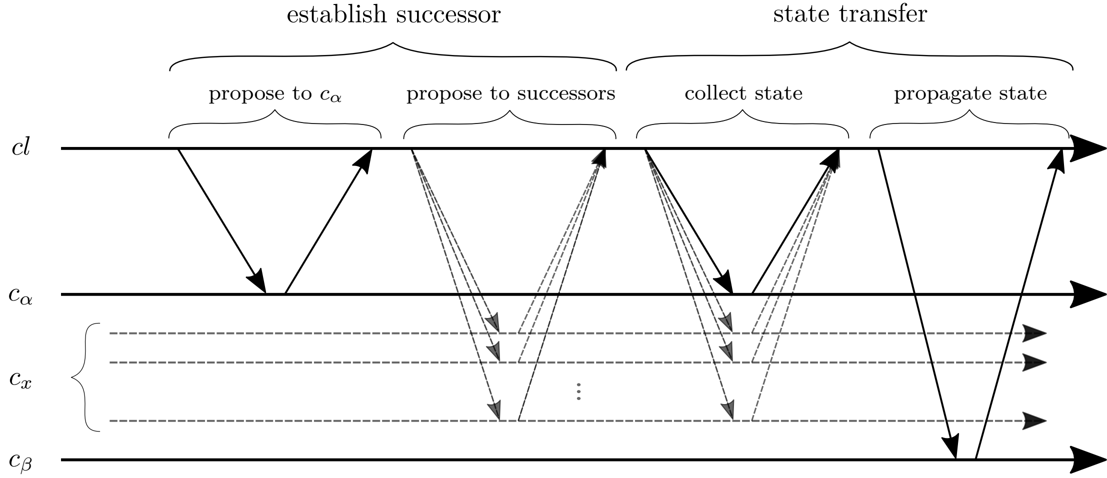
to activate it through the state transfer procedure, resolving the deadlock and ensuring continued availability of the service.

To retain the flexibility of our reconfiguration scheme, the consensus-free approach incorporates the same set of features as its consensus-based counterpart. Hence, it follows that the technique also includes an autonomous reconfiguration component, (L24-L25) in Algorithm 5.4, capable of automatically executing reconfigurations. The autonomous reconfiguration component is triggered if the current reconfiguration fails to successfully install the reconfiguration target according to the user-specified criteria. In an attempt to alleviate the situation and allow for continued availability of the service, the component initiates a new reconfiguration with a reconfiguration target deduced from the currently installed policy. Examples of possible policies include replace faulty servers with non-faulty and available servers or to simply remove the faulty servers, reducing the overall fault-tolerance of the service. The design and development of policies, in addition to the ones already included in Chapter 6, for use by the reconfiguration scheme remains an interesting topic for further work.

A Single Reconfiguration. We now explain how the consensus-free approach performs a single reconfiguration from c_α to c_β, adding an inactive server p_3 to the system running the arbitrary service. The client-side logic of the reconfiguration technique is detailed in Algorithm 5.4, whereas Algorithm 5.5 shows the core logic of the server-side functionality. The scenario is depicted in Figure 5.4, which also visualizes the message pattern of the technique.

The reconfiguration procedure is initiated by a client $cl \in \mathcal{C}$ locally invoking its reconfiguration operation $\textbf{reconf}(\langle +p_3 \rangle, e)$ on c_α where e is a valid epoch larger than any previously used epochs. Following its consensus-based counterpart, the consensus-free technique also enforces a precaution. Ensuring that every invoked non-reconfiguration related quorum call is always executed on the most up-to-date configuration known to the client. Initially, the reconfiguration target is set to equal the client's proposal (L5) and included in the proposal $\langle \text{PRP}\colon c_\beta, c_\alpha \rangle$ to every server in the current configuration c_α.

Upon receiving this proposal, servers in c_α first verify whether c_β is the most up-to-date configuration seen so far, L5 in Algorithm 5.5. If so, servers update their



c_x: intermediate configurations $\{c \in C : c > c_\alpha \land c < c_\beta\}$

Figure 5.4: The message pattern of a reconfiguration operation with the consensus-free reconfiguration technique. A client traverses the configuration DAG, proposing its proposal to every configuration newer than c_α. After c_β is established as the reconfiguration target, the client collects the system's state and propagates it forward to the new configuration. Dashed lines symbolize configurations *cl* learns of during the traversal. Depending on the system's configuration DAG, the set of intermediate configurations may be empty for a given reconfiguration.

local epoch to reflect $c_\beta.epoch$ (L6), implicitly triggering the block release interface as a successor configuration is now established and it is currently inactive. Servers then proceed by adding the proposal c_β to the set of known configurations (L7) and inform the invoker of any relevant successors (L8-L9), helping the reconfiguring clients traverse the configuration DAG. We define a relevant successor to be any configuration where c_α is a predecessor, i.e., the set $\{c \in C\colon\ c.epoch > c_\alpha.epoch\}$. In addition to the set of successors, servers also inform the invoker whether successors have been activated.

A proposal is said to succeed if at least a majority of the servers acknowledges the proposal. *cl* keeps proposing its reconfiguration target to every relevant successor configuration it learns of (L10-L21 in Algorithm 5.4) until either every relevant successor has received the proposal, or it learns of a more up-to-date successor prompting *cl* to switch its reconfiguration target. Should *cl* encounter a more up-to-

date successor, it restarts the reconfiguration procedure with the new target. Thus, ensuring that every configuration previously proposed to, knows that there exists another more up-to-date configuration than the previously proposed target. While *cl* tries to establish its target configuration as the service's next active configuration, it keeps track of every relevant successor it contacts (L19) and records whether the successor has been activated (L13-L14).

After propagating the reconfiguration target to every relevant successor and successfully establishing the target as the service's next active configuration, *cl* proceeds by initiating the state transfer procedure in an attempt to bring the target up-to-date and ready to serve incoming requests. Recall that a configuration is said to be activated once the set state quorum call has forwarded the system's state Φ to the configuration.

The local proxy configuration always reflects an activated configuration. Clients either learn about the configuration from the block release interface, and only activated configurations are released, or they performed the reconfiguration themselves. For the sake of simplicity, assume that the local proxy configuration c_α is the newest configuration in the configuration DAG that is explicitly marked as active. The state delivered to c_α was collected from its predecessors during the state transfer procedure, and reflects the most up-to-date Φ known to the system at the time of invocation. Thus, when propagating the state to c_β, *cl* knows that older configurations $\{c \in C \colon c.epoch < c_\alpha.epoch\}$ contains a possibly obsolete state and do not need to be queried. However, any configurations $\{c \in C \colon c.epoch > c_\alpha.epoch \wedge c.epoch < c_\beta.epoch\}$ encountered during the reconfiguration procedure have may not been activated yet and therefore need to be queried (c_α is the newest activated configuration). *cl* has no way of knowing whether concurrent reconfigurations activated any configurations in the set after *cl* learned of them.

There exists a finite window where *cl* may learn of a successor in its inactive state, the successor activates and processes incoming client requests, updating Φ, before receiving *cl*'s proposal and starts blocking. Consequently, we require the reconfiguring client to query every successor configuration in $\{c \in C \colon c.epoch > c_\alpha.epoch \wedge c.epoch < c_\beta.epoch\}$ for a copy of their Φ (L21-L22). It is safe to execute these quorum calls in parallel, speeding up the procedure. Notice how all collected

Φ are grouped together and sent in a batch to c_β (L24). A possible solution could have been to forward each state as soon as it is available. However, the first time the servers of c_β receive a set state quorum message with an epoch equal to $c_\beta.epoch$, the servers are implicitly activated and ready to process client requests. Therefore, if the servers had received multiple set state quorum calls, the quorum calls could be received in a different order at different servers, inadvertently opening the service to data inconsistencies and other undesired behaviors. If the set quorum call should fail, i.e., *cl* did not receive a quorum of acknowledgements, *cl* triggers the autonomous reconfiguration component in an attempt to remedy the situation (L25).

Concurrent Reconfigurations. When several configurations are initialized concurrently, we need to ensure that every reconfiguration operation eventually converges and targets the same successor configuration. To achieve this, a server keeps track of each configuration it has seen and includes the set of relevant configurations to any client proposing a new configuration. Thus, servers help reconfiguring clients traverse the configuration DAG by informing the reconfiguring clients of any concurrent proposals. To prevent multiple configurations from being active simultaneously, a server only releases a configuration if it has not yet seen a more up-to-date configuration, i.e., the server knows of no other configuration with a higher epoch. Hence, for any pair of concurrent reconfigurations, at least one of the operations will be informed of the other. The client that is informed of the other will therefore either learn of a more up-to-date configuration and change its reconfiguration target, or be reassured that the servers in the current configuration will neither activate nor release the concurrent proposal.

Assume an initial configuration c_i with servers p_1, p_2 and p_3. Further, assume two new configurations c_α and c_β where $c_\beta > c_\alpha$ are proposed concurrently by the two clients $cl_1, cl_2 \in \mathcal{C}$, respectively. If p_1 and p_2 receive the proposal c_β from cl_2 before they receive c_α, they return an empty set of relevant successors to cl_2. cl_2 can then initiate the state transfer procedure, bringing c_β up-to-date and ready to serve incoming requests. When p_1 and p_2 receive the proposal from cl_1, they inform cl_1 of c_β's existence, prompting cl_1 to switch its reconfiguration target. The configuration c_α is never activated. This example is visualized in Figure 5.5a.

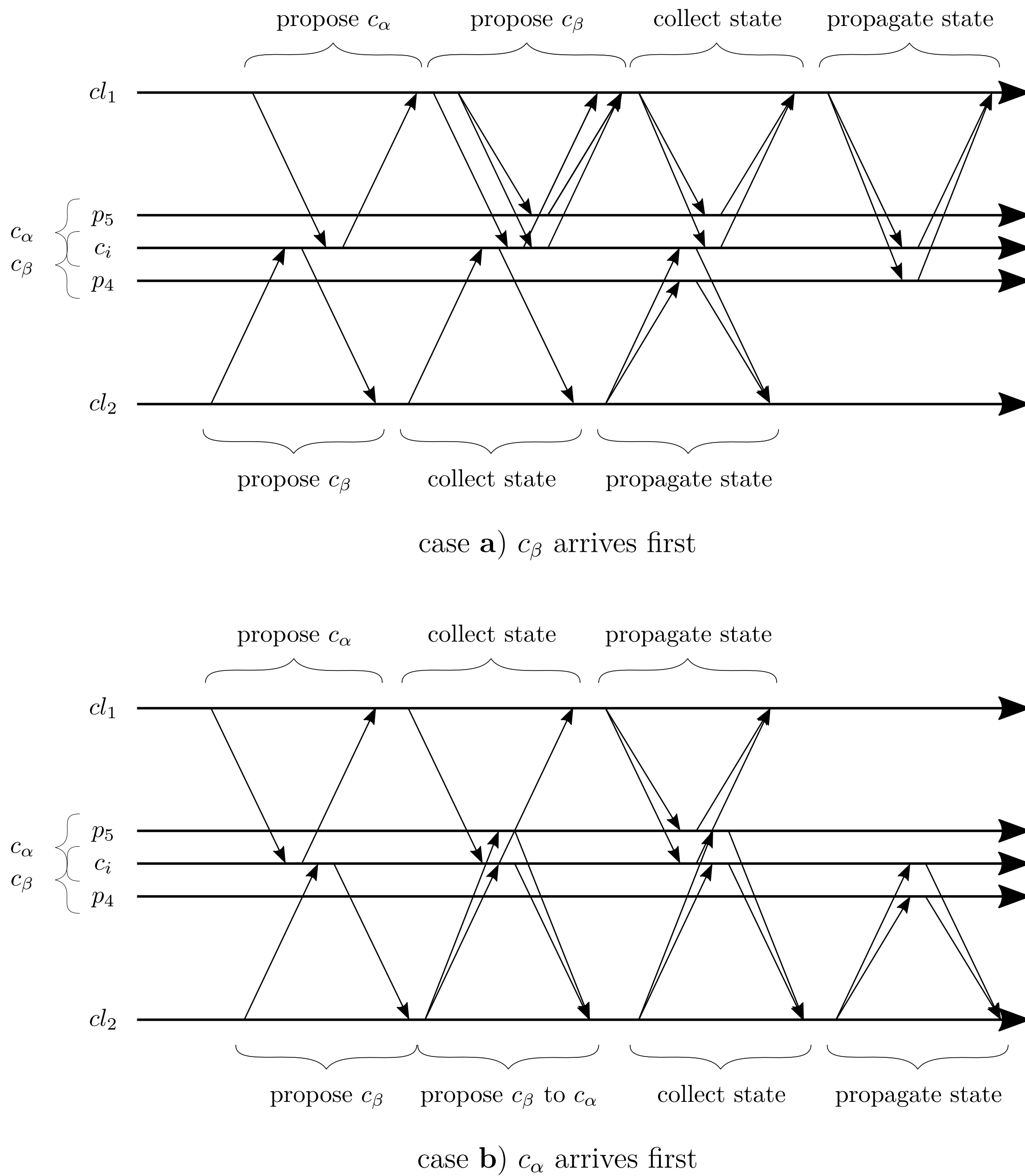


Figure 5.5: In the consensus-free technique, concurrent reconfigurations converge towards an identical configuration. For any two concurrent reconfigurations proposing two distinct configurations c_α and c_β where $c_\beta > c_\alpha$, the technique ensures that eventually c_β is installed and takes precedence over c_α. How this is achieved if the servers of c_i first learns of c_β is depicted in case **a**, whereas case **b** visualizes the technique when c_α arrives first at c_i.

On the other hand, if a majority of c_i were to first receive c_α, as depicted in Figure 5.5b. Then, cl_1 is informed of no successors and initiates the state transfer procedure on c_α. Later, when p_1 and p_2 receives c_β, they inform cl_2 of c_α. If cl_1 completes its state transfer procedure before p_1 and p_2 received cl_2's proposal, c_α is flagged as active and cl_2 only needs to collect the state from c_α. Otherwise, cl_2 proceeds by collecting the state from both c_i and c_α, forwarding them in a single operation to c_β and letting the user-provided logic decide which of the collected states are more up-to-date, and therefore installed. Note that servers in c_i that learn of c_β before receiving the set state quorum call for c_α, will never activate nor release c_α. Even though c_α is now up-to-date, it is not the most up-to-date configuration known to the system. Instead, the servers await the set state quorum call on c_β and subsequently release c_β, assuming that no other clients have proposed a more up-to-date configuration than c_β in the meantime.

5.2 Implementation

In the following sections we present and explain how we have implemented our reconfiguration scheme to work with the Gorums framework [44]. The implementation can be categorized as a proof of concept, and remains to be integrated into the generator logic of the framework. We envision that such an integration extends the client-code generated by Gorums to include the necessary components required for facilitating autonomous reconfiguration of any arbitrary quorum based services defined in protobuf [30] and gRPC [29]. Further, the generator needs to be extended to include additional sever-side code capable of correctly receiving and handling configuration data.

Due to an ongoing discussion [23], debating whether to remove the support for second-order plugins altogether, the generation scheme employed by Gorums has an uncertain future. If removed, it would prompt for a complete rewrite of Gorums' generation logic. Therefore, we decided not to integrate our solution with the current generation scheme, but rather provide a proof of concept that is directly translatable with, and easy to integrate into, the generator.

In Section 5.2.1, we show how the reconfiguration scheme is realized and provide an overview of its components and architecture. We address some practical

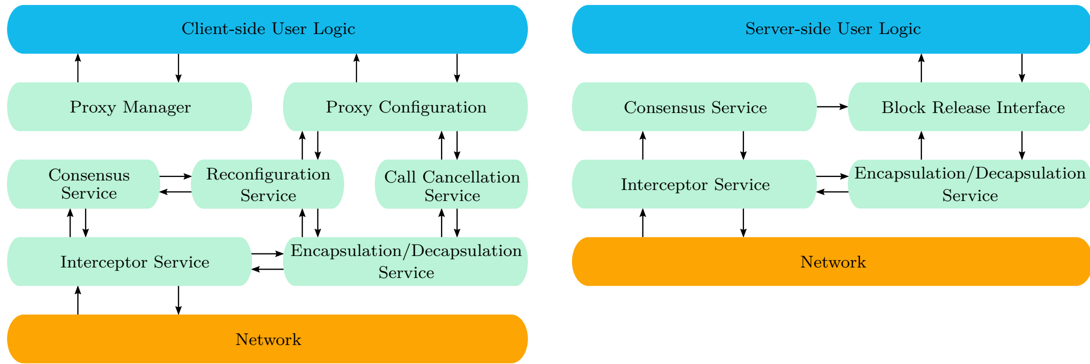


Figure 5.6: The architecture of the Gorums framework extended with reconfiguration capabilities. The figure depicts how the implementation is divided into independent modules and how they interact with each other. Quorum calls invoked by users are registered in the call cancellation service, following its path to the network layer.

problems related to the propagation of quorum information in Section 5.2.2, and in Section 5.2.3 we present an alternative implementation approach that utilizes a layering technique not currently realizable with the current state of the framework.

5.2.1 System Overview and Architecture

Figure 5.6 summarizes our implementation and depicts its architecture. The implementation can be divided into separate self-contained modules that interact with each other. To enhance the flexibility of the implementation, every service module may be replaced with a custom user-provided implementation if desirable, enabling users to integrate custom functionality into the framework to further extend and modify its functionality. At the time of writing, such support is only provided for the interceptor service and the encapsulation/decapsulation service. However, it remains trivial to capture the functionality provided by the remaining services in interfaces by following the pattern used to capture the functionality of the already supported services. Whether it is beneficial to extend this support to new services is questionable and should be looked into further.

The implementation is backwards compatible with the default Gorums framework [44] and gRPC [29]. As for regular Gorums implementations, users first create a manager object and then instantiate a configuration object capable of performing

quorum calls on a group of servers. Reconfiguration capabilities are introduced to the user-defined protobuf service by replacing the manager and configuration objects with their newly introduced proxy counterparts, provided that the solution satisfies the prerequisites imposed by the reconfiguration scheme.

The client-side module overview, with a short introduction to their responsibilities, is as follows:

Proxy Manager. Replaces the original manager object and is the top-level entry point available to users. Its responsibilities include that of the manager as well as option parsing, connection management, and instantiation of proxy configurations.

Proxy Configuration. The proxy configuration is an upper-level object wrapping underlying configuration objects and transposes the frameworks functionality to users, i.e., users invoke quorum calls and reconfigurations on the proxy. It is responsible for registering outgoing quorum calls, initiating reconfigurations on behalf of users, forwarding quorum calls to the underlying configuration object, returning quorum function results to users, and managing the framework's services. The proxy configuration can transparently change the underlying configuration object, targeting a new configuration in the configuration DAG without user awareness.

Encapsulation/Decapsulation Service. Is responsible for transparently wrapping the user-defined RPC messages with the necessary configuration data and to unwrap this configuration data. Unwrapped configuration data is stripped from the received RPC making it unavailable to the user's client-side logic. Stripped data is subsequently registered with the reconfiguration service. Depending on its contents, it might trigger a local reconfiguration.

Call Cancellation Service. Every invoked quorum call is registered in the call cancellation service. The proxy adds quorum calls upon their invocation and they are implicitly removed upon returning. The service provides a handle to the framework, allowing other services to cancel currently registered quorum calls.

Reconfiguration Service. Is responsible for and provides the necessary functionality for performing both local and global reconfigurations. Received configuration data can be passed to the service, which may trigger a local reconfiguration. The implementation provides two reconfiguration services out of the box, one for each reconfiguration technique. Users are free to toggle between the techniques.

Consensus Service. Provides access to a consensus API. It is responsible for initializing new consensus instances on demand, proposing values to instances and providing a handle propagating chosen values to subscribing parties. Currently, the consensus service is only available to reconfiguration operations.

Interceptor Service. The implementation relies on gRPC interceptors to intercept incoming and outgoing RPCs to retrieve, register, and add additional configuration data. Outgoing RPCs are intercepted and augmented with additional metadata. Incoming RPCs are intercepted to retrieve and register the accompanying metadata with the proxy configuration, propagating it to every interested party.

Similarly, the server-side module overview, with an accompanying description, is as follows:

Block Release Interface. The interface functions as an upper-level component linking the different server-side modules together and provides a configuration data store. It is responsible for initializing the server-side functionality and to provide a handle where configuration data can be retrieved and registered. Server-side modules can query the interface to determine the proper course of action for every received RPC, e.g., block the call, release a new configuration, or let it pass through.

Encapsulation/Decapsulation Service. Is responsible for transparently wrapping the user-defined RPC messages with the necessary configuration data and to unwrap this configuration data. On the server-side, metadata is analyzed by the block release interface to decide on the proper course of action.

Unwrapped data is stripped from the RPC, making it unavailable to the user's server-side logic.

Consensus Service. Enables servers to instantiate new consensus instances on demand and provides the server-side functionality of the chosen consensus algorithm. Currently, the consensus service is specialized towards reconfigurations and each instance can choose exactly one successor configuration.

Interceptor Service. The implementation relies on gRPC interceptors to intercept incoming and outgoing RPCs. Incoming RPCs are intercepted to retrieve, register and verify configuration data before forwarding the call to the user-provided service. Outgoing RPCs are intercepted and augmented with additional configuration data.

5.2.2 Quorum Specification

Recall from Chapter 3 that configuration objects in Gorums [44] are immutable and require a user-provided quorum specification upon instantiation. The quorum specification QuorumSpec details the quorum system in use, denoting the quorum size for each operation and can contain additional information necessary to support the user-defined algorithm. Consequently, the quorum system is highly dependant upon the implemented service and subsequently hard to generalize into a one solution fits all.

In practical terms, the QuorumSpec is achieved by generating an interface denoting the signatures of the quorum functions generated for every user-defined RPC. Users are responsible for providing a type satisfying the interface. This allows users to create their own type capturing the necessary information required to satisfy their chosen quorum system, in addition to enabling users to provide their own implementation of every quorum function according to their requirements. During a quorum call, the Gorums runtime simply passes the collected replies to the quorum function by invoking an object satisfying the QuorumSpec interface, to produce a single response capturing the relevant information from the quorum call that the framework returns to the invoker.

```
1  type QuorumSpec interface {
2      /*
3                  .
4                  .
5                  .
6      QuorumFunction(replies []*Reply) (*Reply, bool)
7                  .
8                  .
9                  .
10     */
11
12     // Factory method
13     Update(n int, b ...byte) QuorumSpec
14 }
```

Listing 5.1: The extended quorum specification interface. The interface introduces a new factory method, updating quorum specifications to reflect changes in the configuration. The parameter n denotes the number of servers in the configuration. To support more advanced quorum systems, the method supports an optional field b that can hold arbitrary data to help in the transition.

The current solution is unsatisfactory and incompatible with our proxy configuration. This is because proxy configurations may change the underlying configuration object without user awareness, and therefore can not rely on user-provided quorum specifications for every instantiated configuration. To comply with the proxy configuration, we extend the quorum specification to include an additional factory method, capable of autonomously creating user-defined quorum specifications on demand. Listing 5.1 depicts the extended quorum specification interface. Users provide the initial quorum specification accompanying the initial configuration. Subsequent changes to the active configuration are reflected in the quorum specification by invoking the new factory method. The factory method is user-defined and returns a new quorum specification based on the current quorum specification and configuration data. To support more advanced quorum systems, the factory includes an optional parameter capable of capturing arbitrary information. For example, for a grid quorum system, the optional field can contain encapsulated instructions on how to properly update row and column membership. Moreover, we extend configurations to include the optional quorum specification field and alter the reconfiguration API to accept the optional parameter. If supplied, the optional field follows the configuration data passed transparently to other

clients, helping them update their local quorum specification to correctly reflect the changes in the configuration. Thus, the reconfiguration scheme is capable of providing reconfiguration capabilities to a wide range of quorum based systems, propagating the quorum specifications to active clients in a consistent manner with minimal user intervention.

5.2.3 Alternative Approach

While developing the implementation of our reconfiguration scheme, extending the Gorums framework [44] with reconfiguration capabilities, we encountered some restrictions imposed by Gorums' generation logic. As the framework functions as a second-order plugin to the protobuf [30] compiler, it is incapable of extending the service definitions in the proto file with additional members to the extent necessary to achieve transparent encapsulation/decapsulation of arbitrary configuration data. Ultimately, reducing the number of viable program flows and limiting the possible encapsulation schemes. To comply with, and be compatible with the current generator, the current implementation provides a heuristic approach relying heavily on RPC interception and in exploiting the underlying technologies of gRPC. Instead of altering the service definitions provided by the proto file, we exploit the metadata capabilities of gRPC to include and transmit the required configuration data. Henceforth, we refer to this implementation as the *metadata* based approach.

In the *metadata* based approach, configuration data is included as fields in the HTTP header and relies on interceptors to intercept every RPC to extract, parse, and include the required header fields. The metadata exploit can be viewed as counter-intuitive, as it requires every metadata field to be of type string. Hence, we need to provide custom marshalling and unmarshalling logic to handle non-string types instead of relying on the marshalling and unmarshalling features provided by protobuf, incurring additional performance overhead and redundant code. Moreover, intercepting every RPC call clutters the program flow and results in convoluted code, reducing the comprehensibility of the solution.

Due to the aforementioned concerns, we have developed an alternative implementation approach that utilizes the full power of gRPC [29] and protobuf [30]. Similarly to its *metadata* based counterpart, the *layered* based approach is not

```
// User-defined proto definition
service X {
    rpc RPC(Message) returns (Reply) {}
}

// Details omitted for brevity
message Message {}
message Reply {}

// New proto definition
service gX {
    rpc gRPC(gMessage) returns (gReply) {}
}

message gMessage {
    Message val = 1;
    Conf    conf = 2;
}

message gReply {
    Reply val = 1;
    Conf conf = 2;
}
```

Listing 5.2: The RPC layering technique. User-defined declarations are duplicated and extended to include additional information in quorum calls. The upper-level API accepts and returns the user-defined messages, making the layering technique transparent to the user. The details of the user-defined messages and the content of CONF is omitted for brevity.

yet integrated with Gorums' generator logic. The alternative adopts a layering technique, extending the user-defined messages and RPCs declared in the provided proto file. Every declared RPC is duplicated, changing its message types to new types that capture both the original payload and the additional configuration data we need to propagate transparently. The layering technique is visualized in Listing 5.2. Here, the user-defined service **X** is duplicated into **gX** transmitting GMESSAGE and GREPLY instead of their user-defined counterparts. The new message types contains both the original payload provided by users and the necessary configuration data required by the reconfiguration scheme. The Gorums API exposes only the upper-layer **X** service, accepting a MESSAGE and returning a REPLY. When traversing down the layers of the technique, the received message is encapsulated with additional information and eventually transmitted to the servers by invoking the **gX** RPC. Likewise, collected replies are stripped of additional information

when passing up through the different layers. Each layer of additional information is extracted and registered with its respective service, before forwarding the message to the next layer. When all of the additional information is stripped and registered with their respective services, the user-defined message is passed to the user-provided quorum function, which produces a reply that captures the system's state and is subsequently returned to the invoker. The technique requires an additional layer on the server-side, retrieving and processing the additional information before forwarding the user-defined message to the user-defined gRPC server-side handler. The response retrieved from the server-side handler is augmented with additional information by wrapping it in its layered counterpart before transmitting it back over the wire to the client.

The layering technique is a more favorable approach, providing both well defined and type safe declarations as well as utilizing the marshaling and unmarshalling features provided by protobuf [30]. Further, the technique opens up for a more logical program flow moving up and down through layers instead of the decoupled and convoluted execution flow experienced in the *metadata* based implementation. In Chapter 7, we evaluate the two implementations by comparing their computational overhead. Should [23] result in the removal of the second-order plugin support, making Gorums' generation logic obsolete and in need of a complete rewrite, the layering technique is envisioned to be a viable approach realizable in a stand alone generation scheme. A stand alone generation scheme can accept a proto file, parse it and extend it with the desirable definitions before compiling it with the protobuf compiler.

6 Practical Applications

This chapter illustrates how distributed algorithms can be implemented and extended with reconfiguration capabilities by utilizing the extended Gorums framework presented in Chapter 5. First, we present a data-centric distributed storage algorithm and highlight how the algorithm is implemented in Gorums and extended with reconfiguration capabilities by satisfying our implementation requirements. Then, we showcase the flexibility of the devised scheme and its ease of use by providing an example integration with a process-centric reconfigurable algorithm, illustrating how our data-centric reconfiguration scheme can alleviate developers by fulfilling and replacing the algorithm's reconfiguration component.

6.1 Atomic Register

A register is a storage entity capable of storing exactly one value and provides access to its state through *read* and *write* operations. An atomic register is a register whose operands fulfill atomic [37] semantics, i.e., all operations are linearizable and conforms to a definite order. The ABD algorithm [5] was the first algorithm to provide an atomic register in a distributed setting that is subject to failures [34]. In the ABD algorithm, values are always associated with a logical timestamp and stored in a (value, timestamp)-pair. For any two values, the value with the highest timestamp is said to be greater than the other. To ensure unique timestamps during concurrent operations, processes attach their unique process identifier to

the timestamps they issue. If two timestamps are equal, the process identifier is used as a tiebreaker where the timestamp with the highest identifier takes precedence.

The ABD algorithm assumes a majority quorum system on an immutable set of processes. The quorum system guarantees that values are read from and written to at least a majority of these processes. Both read and write operations follow a similar procedure, performing first a discovery phase that is followed by a propagation phase. During discovery, operations collect the state from a quorum and determine the highest previously used timestamp. The intersection property of a majority quorum system guarantees that any pair of quorums intersect in at least one process and thus a quorum will contain at least one state with the highest known timestamp. A read operation propagates this (value, timestamp)-pair back to a majority, ensuring that successive read operations read this pair, and returns the pair's value to the caller. On the other hand, a write operation increments this timestamp and combines it with its process identifier to create a new, higher, and unique timestamp. This new timestamp is combined with the operation's value into a new (value, timestamp)-pair that is propagated to a majority of the processes. Thus, both operations follow the same message pattern and only differ in the pair transmitted during the propagation phase. The read operation can be optimized to return after the query phase when the read operation is not concurrent with any writes [18], or by providing weaker consistency [56]. Our implementation does not include any optimizations.

RPC Methods. Both the discovery and propagation phase of ABD can be achieved with a single RPC. For the discovery phase, a single RPC capable of returning a (value, timestamp)-pair is sufficient. Likewise, the propagation phase requires a single RPC capable of delivering a (value, timestamp)-pair from a client to a server. The protobuf declarations for these RPCs and their message types are given in Listing 6.1. The Gorums option declared within each RPC informs Gorums' generator logic to extend the gRPC code to include the necessities required for performing quorum calls on said RPCs. In addition to the RPCs required by the algorithm, we define a state transfer service capable of collecting and propagating the algorithm's state in accordance to the implementation requirements of our reconfiguration scheme. Due to the design of ABD, the system's state is easily

```
1  // Algorithm related RPCs
2  service Register {
3      rpc Read(Empty) returns (State)  {
4          option (gorums.qc) = true;
5      }
6      rpc Write(State) returns (Empty) {
7          option (gorums.qc) = true;
8      }
9  }
10
11 message Empty {}
12
13 message State {
14     string val   = 1;
15     Tag    tag   = 2;
16 }
17
18 message Tag {
19     uint64 ts   = 1;
20     uint64 rank = 2;
21 }
22
23 // Additional RPCs required to satisfy the implementation requirements
24 service StateTransfer {
25     rpc GetState(Empty) returns (State) {
26         option (gorums.qc) = true;
27     }
28     rpc SetState(States) returns (Ack)  {
29         option (gorums.qc) = true;
30     }
31 }
32
33 message States {
34     repeated State states = 1;
35 }
36
37 message Ack {
38     bool success = 1;
39     bool updated = 2;
40 }
```

Listing 6.1: The proto definition of the message and RPC declarations necessary to implement the ABD algorithm [5]. The state transfer service and its associated messages and RPCs are required to satisfy the implementation requirements of our reconfiguration scheme.

captured by its RPCs, the state transfer RPCs are almost identical to their algorithm counterparts with the only difference being in the number of possible states captured by the set state RPC message. As previously discussed in Section 5.2, we envision the usage of Gorums options to declare and inform Gorums of the

state transfer RPCs once our reconfiguration scheme is integrated with Gorums' generator logic. For ABD, such an option would enable us to declare the read RPC as its get state procedure, removing the redundant code resulting from declaring a new RPC with identical behaviour.

Quorum Specification. In order to invoke quorum calls on the declared RPCs, we need to provide the framework with a quorum specification object capturing the quorum system in use and its quorum functions. Listing 6.2 depicts a quorum specification designed for use with the protobuf declarations in Listing 6.1, providing quorum functions for both the algorithm's service and the state transfer service in addition to satisfying the implementation requirements. The majority quorum system is captured by a single parameter, denoting the quorum size. To facilitate for autonomously updating the quorum specification to reflect changes in the local configuration, the quorum specification includes a factory method (L5-L9) capable of updating its quorum size parameter. The factory method only requires the number of processes in the new configuration to update the specification, the optional parameter is therefore intentionally excluded.

Due to the similarities between the get state requirements and the discovery phase of ABD, we can reuse the quorum function from the read quorum call for the get state quorum call. The set state quorum function, per our requirements, first awaits replies from all the servers in the new configuration (L33). Once every server has returned a reply, the quorum function inspects their contents (L36-L37) to verify whether the quorum succeeded at a majority, informing the invoker of the result (L38-L39).

State Transfer. As for all gRPC implementations, we are responsible for providing the server-side RPC handlers. Listing 6.3 lists the server-side RPC handlers for the state transfer service defined in Listing 6.1. Upon receiving a get state RPC, a server collects the algorithm's state (L3) and returns it to the caller. The set state RPC may contain multiple states, and as such, servers iterate through the received states (L10), installing them if they fulfill the criteria (L12-L13). Note that even though several states may be installed in consecutive order, the atomic register only stores the final state installed.

```go
type QSpec struct {
	quorumSize int
	size       int
}

// Factory method
func (qs *QSpec) Update(n int, b ...byte) pb.QuorumSpec {
	qsize := int((n / 2) + 1)
	return &QSpec{qsize, n}
}

func (qs *QSpec) ReadQF(replies []*pb.State) (*pb.State, bool) {
	if len(replies) < qs.quorumSize {
		return nil, false
	}
	sort.Sort(ByTag(replies))
	return replies[len(replies)-1], true
}

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

// Quorum functions for the get and set state quorum calls.
func (qs *QSpec) GetStateQF(replies []*pb.State) (*pb.State, bool) {
	return qs.ReadQF(replies)
}

func (qs *QSpec) SetStateQF(replies []*pb.Ack) (*pb.Ack, bool) {
	if len(replies) < qs.size {
		return nil, false
	}
	success := 0
	for _, r := range replies {
	    if r.Success {
	        success++
	    }
	}
	if success < qs.quorumSize {
	    return &pb.Ack{Success: false}, true
	}
	return &pb.Ack{Success: true}, true
}
```

Listing 6.2: The quorum specification object. To comply with the extended Gorums framework, the object contains quorum functions for both the register and the state transfer service along with a factory method, as detailed in Section 5.2.2.

```go
func (r *Register) GetState(ctx context.Context, e *pb.Empty) (*pb.State, error) {
	// locking, logging and error handling omitted for brevity
	resp := &pb.State{Val: r.val, Tag: &pb.Tag{Ts: r.tag.ts, Rank: r.tag.rank}}
	return resp, nil
}

func (r *Register) SetState(ctx context.Context, s *pb.States) (*pb.Ack, error) {
	// locking, logging and error handling omitted for brevity
	resp := &pb.Ack{Success: true}
	for _, ss := range s.GetStates() {
		if less(r, ss) { // r < ss
			r.val = ss.GetVal()
			r.tag = Tag{ts: ss.GetTag().GetTs(), rank: ss.GetTag().GetRank()}
			resp.Updated = true
		}
	}
	return resp, nil
}
```

Listing 6.3: Server-side gRPC handlers for the state transfer RPCs. Notice how the *SetState* method is equipped to handle several states in the received message (L10).

Due to the similarities between the algorithm's phases and the state transfer procedure, the server-side handlers for the two services are almost identical. The read and get state RPCs are identical, whereas the write and set state RPCs only differ in the latter being capable of processing multiple states at once.

By supplying the additional functionality highlighted in Listings 6.1, 6.2, and 6.3, we are able to successfully extend the ABD algorithm with reconfiguration capabilities. The current implementation requires approximately 50 additional lines of code to introduce both consensus-based and consensus-free reconfiguration techniques to the algorithm. Further improvements in the Gorums integration can possibly reduce this complexity further. Utilizing Gorums options to classify state transfer methods would allow us to use the read RPC for both the discovery phase and for collecting the service's state. Additionally, minor tweaking to the write RPC declaration and server-side handler can make it capable of performing both the write tasks and set state tasks. Notice that such a modified write quorum call will either have to change the size of write quorums to include all members (activation requirement) or utilize the correctables option in Gorums and discern between

the two tasks. Thus, for the ABD algorithm, it should be possible to utilize the algorithm's protobuf service for both algorithmic operations and state transfer.

Although the generalized out-of-the-box reconfiguration component can be classified as a heuristic approach, it is successful in providing easy to use reconfiguration primitives at a minimal cost, both in terms of complexity and development time. Thus, the solution succeeds in making it trivial for developers to extend the algorithm with reconfiguration capabilities.

6.2 Vertical Paxos

Recall from Chapter 2 that a state machine can be implemented by executing a sequence of logically separate instances of the Paxos consensus algorithm to choose its commands, instance i choosing the ith command processed by the state machine. Vertical Paxos [40] introduces two Paxos variants to the Paxos suite of algorithms. We examine Vertical Paxos II and simply refer to it as Vertical Paxos for the remainder of this chapter. This particular variant is very much like traditional Paxos [38, 39], but is specialized toward real replicated systems. In both Vertical Paxos and traditional Paxos, the consensus algorithm is executed by four conceptually separate but not necessarily disjoint set of processes: clients, leaders, acceptors, and learners. Leaders propose commands to the instance on behalf of clients, acceptors choose a single proposed command, and learners learn what command has been chosen by the instance. To increase Vertical Paxos applicability in practical replicated systems [40], the variant improves upon traditional Paxos by employing two distinctive differences: (*i*) Vertical Paxos utilizes a read-write quorum system instead of the more traditional majority quorum system, and (*ii*) Vertical Paxos uses an auxiliary configuration master to determine configurations and the leader of every configuration.

While it is possible to let the state machine reconfigure itself, it is often commonplace in practical settings to let a separate configuration master make the reconfiguration decisions [40]. The sequence of independent consensus instances can be visualized as a one-dimensional "horizontal" array. Traditionally, the configuration for a consensus instance in the array is fixed. If the state machine reconfigures itself, it uses the next available consensus instance to choose the suc-

cessor configuration. Let us assume instance j chooses a new configuration, then the changes in the configuration is first reflected (the set of acceptors is different) in consensus instance $j + 1$ and onward. By utilizing an auxiliary master, Vertical Paxos allows the configuration (e.g., set of acceptors) to change within each individual consensus instance. Thus, the array is augmented with a new "vertical" dimension. Configurations change when we move vertically in the now two-dimensional array and remain the same when we move horizontally between instances.

If we let the read quorums in Vertical Paxos be as small as possible, then any single acceptor is a read quorum and the only write quorum is the set of all acceptors. Combining these quorums with the auxiliary configuration master make the replicated state machine implemented by Vertical Paxos tolerate f failures with only $f+1$ acceptors, instead of the $2f+1$ acceptors required without the master [11]. Further, by making the leader one of the acceptors and, upon reconfiguration, choosing the new leader from among the current acceptors. The new leader is by itself a read quorum that already possesses an up-to-date state. Hence, it can perform the state transfer all by itself. If we call this leader the primary and all other acceptors backups, we have a traditional primary-backup system [8].

It is apparent that Vertical Paxos has an increased applicability in practical settings when compared to the ABD algorithm. A replicated state machine is a common and much employed technique in real production systems for ensuring a robust and available service. As such, Vertical Paxos is a prime candidate for showcasing the flexibility and ease of use of our generalized out-of-the-box reconfiguration component, and how the extended framework can alleviate developers by masking complexity and provide time saving abstractions. Additionally, Vertical Paxos has a process-centric design, enabling us to illustrate how our data-centric component can be utilized in this model. To this end, we provide a proof of concept implementation that focuses on integrating a Vertical Paxos consensus instance with the extended Gorums framework. Due to time constraints, the developed implementation is currently incomplete, missing logic that connects the different components among other things. Nonetheless, the Gorums integration is complete and contains all of the logic necessary for satisfying the implementation requirements and for taking full advantage of Gorums' communication abstractions.

The process-centric architecture is achieved by having every machine running Vertical Paxos take the roles of both client and server from the data-centric model. Namely, every machine implements both the client-side and server-side logic required and is capable of both receiving RPCs and invoking RPCs. Such a machine can communicate with all of the other machines by invoking quorum calls on the proxy configuration object, letting each machine's server-side logic answer the call. It is worth pointing out that configuration objects are identical on all machines and include themselves. Thus, a quorum call invoked by machine m on the proxy will trigger an RPC addressed to m. User requests are delivered by new user clients to the set of state machines in a non-quorum call RPC that is targeted directly to one machine. If a machine receiving a client request is not the leader, it forwards the request to the leader. The leader of a configuration is the machine with the lowest identifier. Our generalized out-of-the-box reconfiguration component acts as, and replaces, the configuration master. If a write quorum call fails, the component reconfigures the system, either removing or replacing the faulty machine(s), in accordance to the active policy. For a read one write all quorum system, it is safe for the policy to remove all non-responsive machines without violating protocol properties.

RPC Methods. Following the Paxos consensus algorithm specifications [38], we declare a single RPC for each of the algorithm's phases. For the first phase, the RPC transmits a phase 1a Prepare message and returns a phase 1b Promise message. Likewise, the second phase RPC transmits the second phase equivalents: Accept and Learn, respectively. In addition to the two phase specific RPCs, we declare a third RPC such that users are able to request commands to the state machine. The protobuf declarations for these RPCs and their message types are provided in Listing 6.4. The contents of the message types are omitted for brevity as they are discussed in Section 2.3. Notice the Gorums options declared within the phase specific RPCs. The first option informs Gorums' generator logic to extend the gRPC code to include the necessities required for performing quorum calls on said RPCs. The second option extends the quorum function signature, as detailed in Section 3.3 the option enables us to pass a copy of the originating request to the quorum function.

```
1  // Algorithm related RPCs
2  service VPaxos {
3      rpc Phase1(oneA) returns(oneB) {
4          option (gorums.qc)          = true;
5          option (gorums.qf_with_req) = true;
6      }
7      rpc Phase2(twoA) returns(twoB) {
8          option (gorums.qc)          = true;
9          option (gorums.qf_with_req) = true;
10     }
11     rpc ClientReq(Val) returns (Val) {}
12 }
13
14 // Details omitted for brevity
15 message oneA {}
16 message oneB {}
17 message twoA {}
18 message twoB {}
19 message Empty {}
20
21 // Additional RPCs required to satisfy the implementation requirements
22 service StateTransfer {
23     rpc GetState(Empty) returns (State) {
24         option (gorums.qc) = true;
25     }
26     rpc SetState(States) returns (Ack) {
27         option (gorums.qc) = true;
28     }
29 }
30
31 message State {
32     int   rnd = 1;
33     bytes val = 2;
34 }
35
36 message States {
37     repeated State states = 1;
38 }
39
40 message Ack {
41     bool success = 1;
42     bool updated = 2;
43 }
```

Listing 6.4: The proto definition of the message and RPC declarations necessary to provide a process-centric Vertical Paxos implementation. Notice that the *ClientReq* RPC used by users to deliver requests to the implementation is not a quorum call. Users invoke the RPC directly on one of the active machines. The state transfer service and its associated messages and RPCs are required to satisfy the implementation requirements of our reconfiguration scheme.

To satisfy the prerequisites imposed by our reconfiguration component, we define an additional state transfer service that is capable of collecting and propagating the algorithm's state. Notice how the get state quorum call returns a single state, whereas the set state quorum call is capable of propagating a set of states. Similarly to the phase specific RPCs, the get and set state RPCs are declared as quorum calls. The state of a consensus instance is captured by its safe value and, as such, it is adequate to capture acceptors' (voted round, voted value)-pair to compute the required safe value and propagate it forward to new machines.

Quorum Specification. The quorum specification object is an integral part of Gorums' quorum call abstraction. Not only does the object capture the necessary information required to denote the arbitrary service's quorum system, it also contains quorum functions that give Gorums instructions on how to correctly process and merge collected replies in accordance to the service's requirements. Listing 6.5 depicts a quorum specification object designed for use with the Vertical Paxos protobuf service declaration found in Listing 6.4. The read-write quorum system used by Vertical Paxos is captured by two parameters denoting the size of its respective quorums.

To comply with the implementation requirements of the extended Gorums framework, the quorum specification is extended to include a factory method and quorum functions for the state transfer protobuf service. These quorum functions, as well as the factory method, are detailed in Listing 6.6. The factory method enables the framework to autonomously and transparently update the user-provided quorum specification to reflect changes in the underlying configuration object. Upon the installation of a new local configuration, the method is informed of the total number of servers n in the new configuration and returns a new quorum specification with updated quorum sizes (L5). The read quorum size is always equal to 1, while the single write quorum is the set of all servers. As the factory method only requires the number of machines in the new configuration to correctly update the quorum specification, the optional parameter b is intentionally excluded.

The get state quorum function (L9-L14) awaits upon a read quorum, a single state, and returns it. Even though the leader of a new configuration already possesses an up-to-date state that could be used for state transfer, the extended

```
1  type QSpec struct {
2          readQSize  int
3          writeQSize int
4  }
5
6  func (qs *QSpec) Phase1QF(req *pb.OneA, replies []*pb.OneB) (*pb.OneB, bool) {
7          if len(replies) < qs.readQSize {
8                  return nil, false
9          }
10         reply := &pb.OneB{Rnd: req.GetRnd()}
11         for _, r := range replies {
12                 if r.GetRnd() != reply.GetRnd() {
13                         return nil, false
14                 }
15                 if r.GetVrnd() >= reply.GetVrnd() {
16                         reply.Vrnd = r.GetVrnd()
17                         reply.Vval = r.GetVval()
18                 }
19         }
20         return reply, true
21 }
22
23 func (qs *QSpec) Phase2QF(req *pb.TwoA, replies []*pb.TwoB) (*pb.TwoB, bool) {
24         if len(replies) < qs.writeQSize {
25                 return nil, false
26         }
27         reply := &pb.TwoB{Rnd: req.GetRnd(), Val: req.GetVal()}
28         for _, r := range replies {
29                 if r.GetRnd() != reply.GetRnd() {
30                         return nil, false
31                 }
32         }
33         return reply, true
34 }
```

Listing 6.5: The first part of the quorum specification object. A read-write quorum system is captured by two parameters denoting the size of the read and write quorums. To comply with the extended Gorums framework, the object contains quorum functions for both the Vertical Paxos and the state transfer service along with a factory method, as detailed in Section 5.2.2. The factory method and state transfer quorum functions are displayed in Listing 6.6.

framework invokes a get state quorum call. Our generalized reconfiguration component has no way of knowing whether the machine is already in possession of an up-to-date state and is therefore required to query the previous configuration for a copy of its state. However, recall that the new leader is a member of the previous configuration and that a quorum call invoked on a configuration invokes an RPC to every member of the configuration, including itself. Thus, the single state required

```
35 // Factory method
36 func (qs *QSpec) Update(n int, b ...byte) pb.QuorumSpec {
37         return &QSpec{readQSize: 1, writeQSize: n}
38 }
39
40 // Quorum functions for the get and set state quorum calls.
41 func (qs *QSpec) GetStateQF(replies []*pb.State) (*pb.State, bool) {
42         if len(replies) < qs.readQSize {
43                 return nil, false
44         }
45         return replies[len(replies)-1], true
46 }
47
48 func (qs *QSpec) SetStateQF(replies []*pb.Ack) (*pb.Ack, bool) {
49         if len(replies) < qs.writeQSize {
50                 return nil, false
51         }
52         success := 0
53         for _, r := range replies {
54             if r.Success {
55                 success++
56             }
57         }
58         if success < qs.writeQSize {
59                 return &pb.Ack{Success: false}, true
60         }
61         return &pb.Ack{Success: true}, true
62 }
```

Listing 6.6: The second part of the quorum specification object displays the object's factory method and state transfer quorum functions.

by the get state quorum function can be provided by the RPC addressed to itself and the whole procedure can therefore be viewed as the new leader querying itself for the state. The set state quorum function awaits replies from all of the members in the new configuration (L17-L19). Once the required amount of replies are collected, the quorum function inspect their contents (L20-L25) to verify whether the call succeeded and informs the invoker of the result (L26-L29).

State Transfer. As for all gRPC implementations, we are responsible for providing the server-side RPC handlers. In addition to providing server-side handlers for the Vertical Paxos protobuf service, we need to provide an additional set of handlers for the required state transfer service. The details of the Vertical Paxos server-side handlers are omitted for brevity, but a detailed discussion of their functionality is

```
1  func (v *VPaxos) GetState(ctx context.Context, e *pb.Empty) (*pb.State, error) {
2          // locking, logging and error handling omitted for brevity
3          resp := &pb.State{Rnd: v.rnd, Val: v.vval}
4          return resp, nil
5  }
6
7  func (v *VPaxos) SetState(ctx context.Context, s *pb.States) (*pb.Ack, error) {
8          // locking, logging and error handling omitted for brevity
9          resp := &pb.Ack{Success: true}
10         for _, state := range s.GetStates() {
11                 if state.GetRnd() >= v.rnd {
12                         v.rnd = state.GetRnd()
13                         v.vrnd = state.GetRnd()
14                         v.vval = state.GetVal()
15                         resp.Updated = true
16                 }
17         }
18         return resp, nil
19 }
```

Listing 6.7: Server-side gRPC handlers for the state transfer RPCs. Notice how the *SetState* method is equipped to handle several states in the received message (L10). Even though several states may be installed in consecutive order, the Vertical Paxos instance only stores the final state that was installed.

provided in Section 2.3. The server-side handlers for the state transfer service are detailed in Listing 6.7.

Upon receiving a get state RPC, the handler collects the acceptor's (voted round, voted value)-pair (L3) and returns the collected state to the caller. The set state RPC may contain multiple states, and as such, the handler iterates through the received states (L10), installing them if they fulfill the criteria (L11-L14). Note that even though several states may be installed in consecutive order, the Vertical Paxos instance only stores the final installed state.

After a machine successfully completes the set state RPC handler, the machine's Vertical Paxos instance is bounded by the reported safe value. If the instance was not bounded by any values before the reconfiguration, the reported safe value is a symbolic value indicating that any value is safe for this instance. Otherwise, the safe value continues to be the last voted value as per Paxos specifications.

By providing the additional functionality highlighted in Listings 6.4, 6.5, 6.6, and 6.7, we are able to successfully replace the configuration master in Vertical

Paxos with our generalized out-of-the-box reconfiguration component. Without the complexity of the configuration master and state transfer, Vertical Paxos is mostly reduced to implementing the traditional Paxos protocol. Hence, by providing approximately 50 lines of straightforward code, developers are able to significantly reduce the development time and complexity associated with building fault-tolerant services.

The noticeable similarity between the code required for both extending the ABD algorithm with reconfiguration capabilities and for replacing the configuration master in Vertical Paxos, two completely different scenarios, only strengthens our arguments that the devised reconfiguration component is powerful, flexible, and easy to use.

7
Experimental Evaluation

To provide arbitrary user-defined services with reconfiguration capabilities, we have extended the Gorums framework [44] with our generalized and flexible reconfiguration scheme. The extended framework seamlessly integrates our reconfiguration primitives with the generated logic produced by the framework. In this chapter, we evaluate the extended framework and its newfound reconfiguration techniques, both in terms of performance overhead for common case operations and overhead imposed by reconfigurations on concurrent operations. First, we discuss the experimental setup outlining the experiments. Then, we elaborate on the test scenarios and present the collected metrics and results, further discussing our findings and their implications.

7.1 Experimental Setup

All experiments are performed in a wide area (WAN) setting and run on virtual machines hosted on the Google Compute Engine [28] service component of the Google Cloud Platform (GCP) [27]. To ensure a WAN environment, we evenly distribute virtual machines to the four available data centres located in Europe [26]: United Kingdom, Netherlands, Belgium, and Germany. Moreover, virtual machines hosted within the same data centre are allocated to different availability zones to guarantee that they are run on separate physical machines.

For our evaluation, we utilize the unoptimized ABD algorithm [5] implementa-

Table 7.1: Specifications for client and server virtual machines [24].

	Model	vCPU	Memory (GiB)	Network Tier
Client	n1-standard-2	2	7.5	Premium (Gigabit)
Server	n1-standard-2	2	7.5	Premium (Gigabit)

tion presented in Section 6.1 as the top level algorithm extended with reconfiguration capabilities, i.e., the ABD algorithm serves the role of the user-defined service. Unless otherwise noted, every experiment is executed with 5 servers and 16 clients, each invoking 1000 *write* operations with a payload size of 128 kB. Recall that *write* operations in ABD consists of both a discovery and a propagation phase. Thus, for every *write* operation invoked by a client, the client needs to first collect the system's state to discover the most up-to-date timestamp before propagating the payload in the newly issued (value, timestamp)-pair. Hence, the 1000 *write* operations issued by a client is in fact 2000 quorum call invocations each with a payload of 128 kB.

Each of the five servers are running on their own dedicated *n1-standard-2* virtual machine, whereas the 16 clients are divided into groups of four and run on four additional instances of *n1-standard-2* virtual machines. The four client virtual machines are evenly distributed among the four available data centres. Similarly, the five servers are divided upon the data centres, with one data center hosting two servers to accommodate the fifth instalment. All virtual machines are running Linux 4.4.0, uses Go version 1.9.3 for code compilation, and are interlinked with Google's premium [25] Gigabit network tier, taking full advantage of their global network. The hardware capabilities of the virtual machine instances are summarized in Table 7.1. Preliminary testing reveals that this setup utilizes approximately $60\% - 90\%$ and $70\% - 80\%$ of the resources available to servers and clients, respectively.

To provide a notion of the network latency experienced in our WAN environment, we have performed some initial latency related testing. Table 7.2 summarizes these results and presents the mean round trip time experienced between the client virtual machines and each of the server virtual machines. The results provide both an indication of the experienced network overhead as well as helping us visualize our

Table 7.2: Mean network latencies in milliseconds between the machines in the WAN environment.

Client	Server 1	Server 2	Server 3	Server 4	Server 5
1-4	0.474	7.303	7.793	9.877	7.309
5-8	7.487	0.501	7.502	6.513	0.636
9-12	9.944	6.370	13.669	0.588	7.082
13-16	7.474	7.562	0.691	13.659	7.559

WAN environment. Round trip times of less than 1 ms indicates that the virtual machines are co-located within the same data centre and thus shares a local area (LAN) connection.

The solution's deadlock detection mechanism is disabled for all implementations for the duration of the experiments. The deadlock detection mechanism detects and resolves the single point of failure scenario described in Section 5.1.2. As we do not perform any failure scenarios nor test for such behaviour, we have disabled the mechanism to prevent false positives from disturbing system operations and influencing the recorded metrics.

7.2 Common Case Operations

To better understand the baseline cost of adopting our reconfiguration scheme, we first examine the performance overhead incurred on common case operations. That is, we are interested in assessing the added cost in terms of latency experienced by system operations in a stable environment without any concurrent reconfiguration operations. To have a meaningful basis of comparison for the evaluation of both of our implementation alternatives, we have developed a third implementation of the ABD algorithm [5] in the unaltered regular Gorums framework [44]. We refer to this variant as the *plain* implementation when discussing it and comparing it against our *metadata* based and *layered* based implementations.

The experiment follows the specifications given in Section 7.1. We use 5 servers as this is the minimal requirement capable of providing a fault-tolerance of two

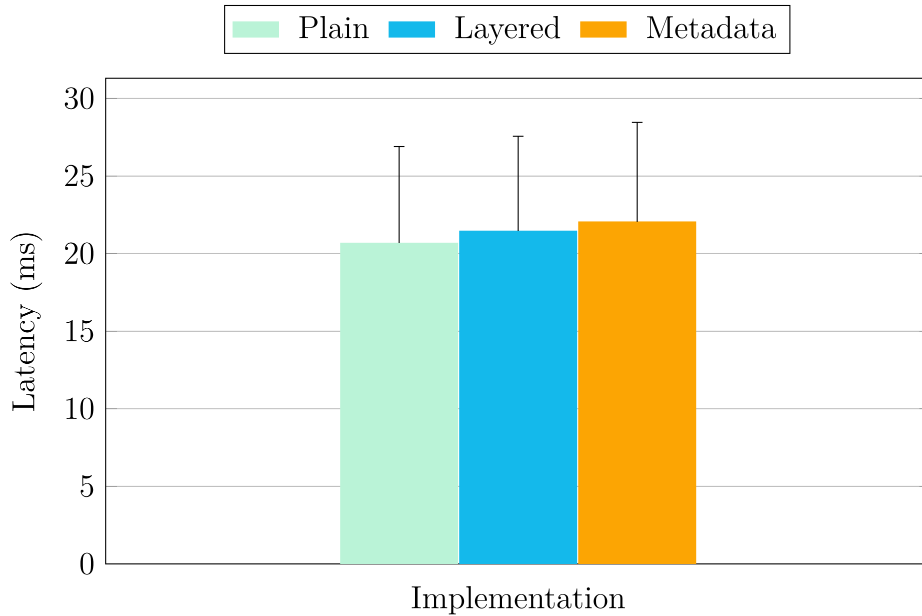


Figure 7.1: Mean latency and 95th percentile (in milliseconds) of a complete *write* operation in the ABD algorithm [5], as recorded by our three different implementations. The *plain* variant is not extended with reconfiguration capabilities. Both *layered* and *metadata* variants have reconfiguration capabilities, and only differs in their implementation technique, as detailed in Section 5.2.3.

in ABD's majority quorum system. All 16 clients communicate with the servers using RPCs over TCP connections. To prevent connection establishment and management from interfering with the collected metrics, socket creation and other network related necessities are performed before we begin the experiment. The experiment is repeated a total of 10 times for each implementation.

Collected latencies reflect the total elapsed time from a client initiates the write operation, to the operation returns. More specifically, it does not measure the time for each RPC, but rather for the operation as a whole. Note that each operation contains two quorum calls that first return after receiving replies from a minimum of 3 servers.

The evaluation results are summarized in Table 7.3 and visualized in Figure 7.1. The figure depicts the mean latency and the 95th percentile recorded by the different implementations. In an attempt to remove network anomalies, the collected datasets are trimmed, removing 0.5% of the top and bottom records. As expected,

Table 7.3: Time to perform *write* operations in the ABD algorithm [5] in a stable non-reconfiguring environment. All times are recorded in milliseconds.

	Write times (ms)			
Implementation	Mean	Stdev	Min	Max
Plain	20.674	3.240	14.830	33.810
Layered	21.451	3.253	15.740	33.520
Metadata	22.042	3.420	16.050	34.330

the *plain* variant outperforms the other two. The *plain* implementation does not provide configuration capabilities, and as such, does not need to transmit additional configuration data in every invoked RPC. Consequently, the *plain* variant transmits smaller packets, incurring less computational overhead from serialization/deserialization and experiences less networking delay. Between the *layered* and the *metadata* implementations, it is apparent from the figure that the former implementation outperforms the latter. Both implementations incorporate our reconfiguration scheme and consequently transmit identical configuration data in every instantiated RPC. These findings are in line with what is to be expected. It is reasonable to assume that the serialization and deserialization provided by protobuf [30] is more effective than our custom metadata equivalent.

Overall, the additional computational cost incurred on common case operations is roughly 4% and 7% for the *layered* and *metadata* variants, respectively. For practical systems, this seems like a small price to pay for easily introducing reconfiguration capabilities into the service and should not be so substantial as to deter its use nor affect user-experience. However, it should be noted that the cost is not negligible and if a system does not require reconfiguration capabilities, it is better of by excluding the solution altogether.

7.3 Reconfiguration Overhead

In our next experiment, we are interested in assessing the overhead reconfiguration operations impose on concurrent system operations and to evaluate the performance

of our data-centric Paxos variant. To this end, we extend the experimental setup and test case from Section 7.2 with a new identical *n1-standard-2* virtual machine hosting the reconfiguring clients, for a total of 5 servers and 5 client virtual machines.

At one point during the experiment, we start 1, 2, or 3 clients, each issuing a global reconfiguration. Each reconfiguration operation proposes to remove an arbitrary server from the initial configuration, reducing the number of active servers from 5 to 4. Note that majority quorum systems on 4 and 5 processes share the same quorum size, requiring a minimum of 3 replies to obtain a majority. Thus, after the reconfiguration is complete, quorum calls invoked by clients as part of *write* operations continue to require a minimum of 3 replies. We repeat the experiment 10 times for each combination of reconfiguration technique and number of concurrent reconfigurations, bringing the total up to 60 independent runs. All experiments are run on the *metadata* based implementation.

For the consensus-based technique, we record the *chosen time* for its consensus instance to examine the performance of our data-centric Paxos variant. We define the chosen time to be the duration from the client (that got its proposal chosen by consensus) proposes a value to the consensus instance, until it is informed of the chosen value. In this scenario, collected latencies reflect the total elapsed time from a client initiates the write operation, to the operation returns. We are only interested in the latencies for operations that are concurrent to and thus influenced by the reconfiguration operations. At some point during the write operation, either in the discovery or propagation phase, the invoked RPC is blocked by the servers until the concurrent reconfiguration completes and activates the successor configuration. Once the successor configuration is established and activated, the RPC is released, returning the successor configuration to the invoking client. Upon receiving the successor configuration, our solution performs a local reconfiguration at the client, installing the successor locally, and retransmits the aborted RPC. With the system stabilizing at the new successor configuration, the *write* operation is allowed to proceed and eventually complete, storing its elapsed duration in memory for the remainder of the experiment.

Figure 7.2 visualizes our findings, showing the mean and 95th percentile for both the chosen times and the *write* latencies experienced for operations concurrent to 1-3 reconfigurations. Further, the results are summarized in Tables 7.4 and 7.5.

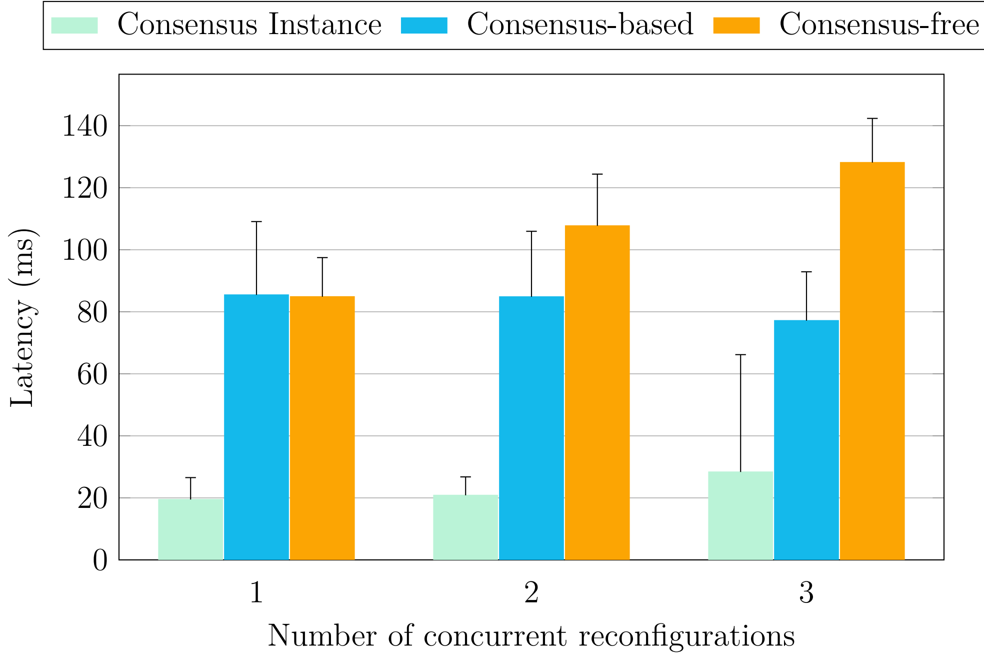


Figure 7.2: Mean latency and 95th percentile (in milliseconds) of consensus chosen times for 1-3 concurrent proposals and for *write* operations in the ABD algorithm [5] that are concurrent to 1-3 global reconfiguration operations for both the consensus-based and consensus-free technique, as detailed in Sections 5.1.2 and 5.1.3, respectively.

In the scenario with only a single proposal to the consensus instance, we find that it takes approximately 20 milliseconds for the proposing client to establish a successor and get its proposal chosen by consensus. The data-centric Paxos variant introduced in this thesis contains three phases, and therefore requires three round trips between the proposing client and a majority of the servers. Keeping this in mind, and taking the mean round trip times experienced in our WAN environment (Table 7.2) into consideration, we can contribute most of this duration to waiting and network overhead. Hence, the results are promising and indicate that our data-centric Paxos variant can be highly efficient, with its cost being directly proportional to the system's round trip times and network architecture. Likewise, we notice similar results for the scenario with two concurrent proposals to the consensus instance. This is reasonable and explained by the fact that for any two concurrent proposals, only a single proposal wins the round. The other

Table 7.4: Chosen times in milliseconds for concurrent proposals.

	Chosen times (ms)			
Num concurrent	Mean	Stdev	Min	Max
1	19.462	4.606	17.040	32.350
2	20.826	3.447	17.410	29.640
3	28.336	26.054	17.490	102.380

proposal backs off and tries again at a later point of time, giving the winner time to try to get its proposal chosen by consensus. During evaluation, the exponential back off policy is set rather relaxed, starting at 50 milliseconds, giving the winner adequate time to proceed and get its proposition chosen before the other client retries the algorithm. For three concurrent proposals to the consensus instance, we notice a slight performance degradation resulting in clients using on average 28 milliseconds to establish a successor configuration. This is to be expected, as it is not unreasonable to assume that higher contention increases the likelihood of clients disrupting each others' operations. However, due to the exponential back off policy, a single client is eventually able to establish a successor configuration within reasonable time.

When examining all of the consensus-based experiments, we interestingly find that the mean latency spike experienced by concurrent system operations remains fairly constant at approximately 82 milliseconds. Recall that the block release interface first starts blocking when a successor configuration is established and available, i.e., after the first phase of the consensus-based reconfiguration technique is complete. Ergo, the interface only blocks for the duration of the state transfer procedure. Moreover, by the power of consensus, the consensus-based reconfiguration technique guarantees that there is at most one successor configuration. Consequently, the technique does not experience any significant additional overhead from concurrent reconfigurations. Every reconfiguring client is informed of the same identical successor and tries to install the successor configuration by collecting the system's state and forwarding it to the successor. The first client that succeeds, releases the successor configuration to awaiting clients. However, as

Table 7.5: Time to perform *write* operations in the ABD algorithm [5] that are concurrent to ongoing reconfigurations. All times are recorded in milliseconds.

	Write times (ms)			
Num concurrent	Mean	Stdev	Min	Max
Consensus-based:				
1	85.429	13.956	61.000	149.080
2	84.815	13.456	60.660	140.830
3	77.141	10.919	61.910	142.090
Consensus-free:				
1	84.842	8.636	66.560	139.060
2	107.699	16.377	72.160	238.950
3	128.108	9.044	89.280	150.850

previously discussed, concurrent reconfigurations do affect the time used by our consensus instance to agree upon a single value. Although, as this increased cost is concurrent to common case operations and does not trigger the blocking behaviour of the interface, it is neither visible to nor affects non-reconfiguring clients.

The cost experienced by concurrent common case operation can be broken down into five distinctive parts: performing the *write* operation of the ABD algorithm before the system blocks, waiting for the successor to be released, installing the successor locally, retransmitting the blocked and aborted RPC to the new configuration, and performing the remainder of ABD's *write* operation. Unfortunately, due to time constraints and limited funding, we do not have enough details in our logs to accurately time these periods with a high degree of confidence. An interesting topic for further work would be to introduce tracing to the code and create a better visualization of our solution and the cost of each of the aforementioned parts. What we do know, by comparing these results with the metrics recorded for the stable environment, is that the consensus-based reconfiguration technique introduces an additional cost of approximately 60 milliseconds to concurrent common case operations.

A single reconfiguration in the consensus-free technique has a lot in common

with the consensus-based approach. As there is only a single reconfiguring client, the consensus-free technique produces exactly one successor configuration and the subsequent state transfer procedure only needs to collect the state from a single configuration. In terms of the block release interface, servers start blocking common case operations once they receive the successor configuration. Due to there only being a single successor, the reconfiguring client is not informed of any conflicting configurations it needs to contact and can therefore initiate the state transfer procedure, collecting the state from the current configuration and propagating it to the reconfiguration target. Hence, it does not come as a surprise that this scenario performs on par with the consensus-based technique in terms of overhead imposed on concurrent common case operations. From a reconfiguring client's perspective, the consensus-free technique with a single reconfiguration outperforms the consensus-based approach, as the reconfiguring client in the latter case needs to await the consensus instance to be informed of the reconfiguration target.

For 2 and 3 concurrent reconfigurations, we notice an increase in the overhead imposed on concurrent common case operations, growing with approximately 21 milliseconds per concurrent reconfiguration. In the consensus-free technique, servers block as soon as they receive a successor configuration. Consequently, in addition to waiting for the state transfer procedure to complete and release the successor, non-reconfiguring clients need to also wait for the concurrent reconfigurations to converge and establish the next successor. Moreover, with multiple successors, reconfiguring clients are required to collect the state from multiple configurations, increasing the overall time for completing the state transfer procedure and inadvertently influencing the blocking duration negatively. Therefore, it is evident that common case operations favor the consensus-based technique, as it imposes the least overhead on concurrent operations of our two techniques.

When considering the frequency of reconfigurations in practical systems, where reconfigurations are infrequent and far in between, both techniques show promising results and complete within reasonable time. In a controlled environment where reconfiguration capabilities are restricted to a few trustworthy clients, we believe the consensus-free technique is capable of competing with its consensus-based counterpart. Note that the overhead imposed on concurrent operations by the consensus-based technique is mostly related to the state transfer procedure. Like-

wise, the state transfer procedure remains a major part of the overhead produced by the consensus-free technique. As a consequence, the reconfiguration results are closely related to the performance of the state transfer operation. For this reason the results presented in this section are only representative for the ABD algorithm [5] with a 128 kB state. Different algorithms introducing different states, with their own metrics, will experience different reconfiguration overheads than the overheads reported in this chapter.

8

Conclusion and Further Work

This chapter concludes the thesis and suggests some interesting topics for further work.

8.1 Conclusion

The overall goal of this thesis was to examine and extend the Gorums framework with easy to use, available, and adaptable reconfiguration abstractions. To this end, we developed a generalized out-of-the-box reconfiguration component with design goals of (*i*) minimal performance overhead during common case operations, and (*ii*) providing sufficient adaptable abstractions for easily introducing reconfiguration capabilities to arbitrary services.

During our work of integrating the reconfiguration component with the logic produced by Gorums, we found no satisfactory method for transparently augmenting RPCs with the required configuration data. Consequently, we proposed two alternative approaches for realizing our devised reconfiguration scheme. The *metadata* based approach is compatible with the current generator logic of Gorums and exploits the underlying technologies to transmit the required configuration data as metadata fields in the HTTP header of gRPC. Alternatively, we propose a better and less convoluted *layered* approach that augments the originating RPC by wrapping it in a new RPC containing the necessary configuration data. Currently, the *layered* alternative is not a viable option for the framework as it is not compatible

with Gorums' generator logic. However, depending on the future of the framework, the *layered* alternative is superior and the recommended approach for realizing the reconfiguration component should the framework rewrite its generator logic.

Experimental evaluation revealed that both approaches fulfill our first design goal, imposing a minimal overhead on common case operations in a stable non-reconfiguring environment. In line with our expectations, the *layered* approach outperforms its *metadata* counterpart by a tiny fraction, which we contribute to its utilization of protobuf's superior serialization/deserialization techniques. Moreover, we integrated two distributed algorithms with the extended Gorums framework. The first algorithm was extended with reconfiguration capabilities, whereas for the second algorithm, our data-centric reconfiguration scheme successfully replaced the algorithm's process-centric reconfiguration component. Both algorithms required only minor adaptions and almost identical trivial code to incorporate our out-of-the-box reconfiguration component. Highlighting our out-of-the-box reconfiguration component's ease of use.

For providing reconfiguration capabilities to services operating in environments unfit for consensus, we have, in addition to the consensus-based reconfiguration technique, developed a consensus-free reconfiguration technique. Experimental evaluation of the techniques show promising results. Indicating that both techniques are able to successfully complete within reasonable time. While the consensus-free technique performs on par with its consensus-based counterpart for a single reconfiguration, its cost increases linearly for every concurrent reconfiguration operation. This is in contrast to the consensus-based technique, where the cost remains constant.

Although our generalized out-of-the-box reconfiguration component can be classified as a heuristic approach, it will never be as efficient as the specialized state-of-the-art reconfiguration techniques, it can easily introduce reconfiguration capabilities to arbitrary services that complete within reasonable time. Considering that the most important part for ensuring continued availability of a replicated service, and to retain the service's fault-tolerance in the face of failures, is the ability to perform reconfigurations. Further, due to the infrequency of reconfigurations in practical systems, it is not necessarily important to optimize the procedure. Thus, we consider it safe to claim that our devised reconfiguration component

satisfies the second design goal and, consequently, further helps in alleviating developers of the complex, tedious, and time-consuming processes of building truly fault-tolerant services. The reconfiguration component masks the intricacies of reconfiguration and instead presents developers with an easy to use and intuitive reconfiguration API.

8.2 Further Work

In this section, we provide some thoughts on further directions and interesting topics that continue on the work presented in this thesis. In addition, we suggest some optimizations that can be applied to the devised scheme to further improve upon its performance.

Further Experiments The evaluation conducted as part of this thesis uses a basic single failure scenario in the performed experiments. A more complex failure scenario, e.g., network partition, should be tested in experiments for evaluating the presented reconfiguration component further.

Autonomous Reconfiguration Component Currently, the extended Gorums framework only performs reconfigurations after being explicitly triggered by a client with reconfiguration permissions. Therefore, in order to provide autonomous reconfigurations of the service, it would be interesting to develop a new monitoring component that can initiate reconfigurations on behalf of clients in accordance to a given set of criteria. The Gorums framework already collects and exposes latency and error related metrics to clients and it would therefore be trivial to act upon this information to autonomously alter the configuration through the reconfiguration API.

Reconfiguration Policies Reconfiguration policies as a last resort to resolve a potentially dangerous situation that would render the system unavailable is an interesting and difficult topic worth investigating. The topic is further complicated by the fact that different quorum systems impose different requirements on policies, and policies that are safe for a given system is not necessarily safe for another quorum system. Currently, we provide a single

reconfiguration policy that is safe for read-write quorum system. When invoked, the policy attempts to relieve the system by temporarily lowering its fault-tolerance and requires user intervention to restore the system's original fault-tolerance. Developing and designing new policies accommodating different use-cases and different quorum systems, as well as improving the framework's support for policies, are all interesting tasks that would further improve the reconfiguration component.

Explicit Activation RPC One could argue that the current reconfiguration component places to much complexity and responsibility on the developer. Incorrect implementations of the set state RPC quorum function and server-side handler can have severe consequences for the liveness and fault-tolerance of the implemented service. Consequently, it would be interesting to perform a study investigating whether it could be more beneficial to move the additional responsibilities of the set state RPC to a new dedicated activation RPC. The new activation RPC replaces the implicit responsibilities of the set state RPC and explicitly activates new configurations once the state transfer is complete, prompting machines to release the new configuration. Introducing such an RPC also comes with the added benefits of making it easier to reason about the component's correctness and reduces the number of machines that need to receive the set state RPC, which might be a very costly operation.

Optimizing Global Reconfigurations We found that adopting the technique of blocking system operations for the duration of the state transfer procedure to the data-centric model is problematic. In the data-centric model, the technique creates a critical single point of failure that could render the complete system unavailable and leave it blocking indefinitely. Currently, we solve this problem by equipping clients with a deadlock detection mechanism and allow concurrent reconfigurations to initiate concurrent state transfers. It would be interesting to investigate whether we could resolve this single point of failure in a more efficient manner, e.g., by utilizing a lease-based approach [46]. State transfer is the most costly operation in reconfigurations, and reducing the number of concurrent, most likely superfluous, state transfer procedures

could greatly improve the performance of the reconfiguration operation as well as remove unnecessary contention and network load.

Moreover, as state transfers are such a costly operation, it would be advisable to improve the set state RPC. Instead of transmitting the state to every member of the new configuration, we could utilize a per server map function to only transmit the state to new members. Omitting the state in the messages targeted to members of both the new configuration and its predecessor.

Attachments

A.1 Source Code

A copy of the full coding work done for this thesis is embedded within this thesis. The embedded source code contains both implementation approaches, the two distributed algorithms and their integration with the extended framework, and all the code used to perform the experimental evaluation.

- Embedded: source_code.7z

A.2 Printer-Friendly Version of Thesis

A printer-friendly version of this thesis is embedded within this thesis. The printer friendly version includes additional blank pages, no formal front page, and no attachments appendix.

- Embedded: Frausing, Tor Christian-print.pdf

A.3 Raw Experimental Data

The complete raw experimental data collected during evaluation is embedded within this thesis. The raw data is categorized per experiment and located in their respective folders. Filenames follows the format: <role>_<client id>_TEST<run number>. Data within the files are formatted according to the csv standard, using a whitespace as delimiter. Rows contain the operation performed, starting time of the operation, time of completion, and the elapsed time of the operation.

– Embedded: experimental_data.7z

B

Complete Experimental Data

This appendix contains the complete experimental data for both the common case operation experiment and the reconfiguration overhead experiment in Chapter 7. The data is aggregated and presented in the form of summary statistics. The raw data can be found embedded within this thesis in Section A.3.

Table B.1: Write times in a stable non-reconfiguring environment for the three different implementations.

	Write times (ms)			
Implementation	Mean	Stdev	Min	Max
Plain	20.674	3.240	14.830	33.810
Layered	21.451	3.253	15.740	33.520
Metadata	22.042	3.420	16.050	34.330

Table B.2: Chosen times for concurrent proposals to the consensus instance.

	Chosen times (ms)			
Num concurrent	Mean	Stdev	Min	Max
1	19.462	4.606	17.040	32.350
2	20.826	3.447	17.410	29.640
3	28.336	26.054	17.490	102.380

Table B.3: Write times influenced by concurrent consensus-based reconfigurations.

	Write times (ms)			
Num concurrent	Mean	Stdev	Min	Max
1	85.429	13.956	61.000	149.080
2	84.815	13.456	60.660	140.830
3	77.141	10.919	61.910	142.090

Table B.4: Write times influenced by concurrent consensus-free reconfigurations.

	Write times (ms)			
Num concurrent	Mean	Stdev	Min	Max
1	84.842	8.636	66.560	139.060
2	107.699	16.377	72.160	238.950
3	128.108	9.044	89.280	150.850

List of Algorithms

3.1 Default Quorum Function Signature 19
3.2 Data-centric Paxos Phase 1b Quorum Function 20
3.3 Quorum Function Signature with Initial Request 22
3.4 Quorum Function Signature with Custom Return Type 23
3.5 Quorum Function with Custom Return Type 24
5.1 Data-centric Consensus-based Reconfiguration 45
5.2 Data-centric Paxos (Client) 48
5.3 Data-centric Paxos (Server) 49
5.4 Data-centric Consensus-free Reconfiguration (Client) 55
5.5 Data-centric Consensus-free Reconfiguration (Server) 55

List of Figures

2.1 Reconfiguration Procedure 9
2.2 (Single-decree) Paxos 12
2.3 Data-centric Paxos 15

3.1 Quorum Call 18
3.2 Quorum Function 19
3.3 Quorum Call with Per Node Arguments 22

4.1 Consensus-based Configuration DAG 32
4.2 Consensus-free Configuration DAG 35

5.1 Reconfiguration Interface 41
5.2 A Single Reconfiguration (Consensus-based) 50
5.3 Concurrent Reconfigurations (Consensus-based) 52
5.4 A Single Reconfigurations (Consensus-free) 57
5.5 Concurrent Reconfigurations (Consensus-free) 60
5.6 System Architecture 62

7.1 Common Case Results 88
7.2 Reconfiguration Overhead 91

List of Listings

5.1 Quorum Specification 66
5.2 RPC Layering Technique 68
6.1 Atomic Register RPC Methods 72
6.2 Atomic Register Quorum Specification 74
6.3 Atomic Register Server-side State Transfer Methods 75
6.4 Vertical Paxos RPC Methods 79
6.5 Vertical Paxos Quorum Specification Part 1 81
6.6 Vertical Paxos Quorum Specification Part 2 82
6.7 Vertical Paxos Server-side State Transfer Methods 83

List of Tables

7.1 Virtual Machine Specifications 86
7.2 Network Round Trip Times 87
7.3 Common Case Operations 89
7.4 Chosen Times for Concurrent Proposals 92
7.5 Common Case Operations Concurrent to Reconfigurations 93

B.1 Write Times iii
B.2 Chosen Times iv
B.3 Consensus-based Reconfigurations iv
B.4 Consensus-free Reconfigurations iv

Bibliography

[1] M. K. Aguilera, R. Janakiraman, and L. Xu, “Using erasure codes efficiently for storage in a distributed system”, in *2005 International Conference on Dependable Systems and Networks (DSN’05)*, Jun. 2005, pp. 336–345. DOI: 10.1109/DSN.2005.96.

[2] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, “Dynamic atomic storage without consensus”, *J. ACM*, vol. 58, no. 2, 7:1–7:32, Apr. 2011, ISSN: 0004-5411. DOI: 10.1145/1944345.1944348. [Online]. Available: http://doi.acm.org/10.1145/1944345.1944348.

[3] E. Alchieri, A. Bessani, F. Greve, and J. da Silva Fraga, “Efficient and modular consensus-free reconfiguration for fault-tolerant storage”, *CoRR*, vol. abs/1607.05344, 2016. arXiv: 1607.05344. [Online]. Available: http://arxiv.org/abs/1607.05344.

[4] J. W. Anderson, H. Meling, A. Rasmussen, A. Vahdat, and K. Marzullo, “Local recovery for high availability in strongly consistent cloud services”, *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 2, pp. 172–184, Mar. 2017, ISSN: 1545-5971. DOI: 10.1109/TDSC.2015.2443781.

[5] H. Attiya, A. Bar-Noy, and D. Dolev, “Sharing memory robustly in message-passing systems”, *J. ACM*, vol. 42, no. 1, pp. 124–142, Jan. 1995, ISSN:

0004-5411. DOI: 10.1145/200836.200869. [Online]. Available: http://doi.acm.org/10.1145/200836.200869.

[6] R. Baldoni, S. Bonomi, A. M. Kermarrec, and M. Raynal, "Implementing a register in a dynamic distributed system", in *2009 29th IEEE International Conference on Distributed Computing Systems*, Jun. 2009, pp. 639–647. DOI: 10.1109/ICDCS.2009.46.

[7] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls", *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, Feb. 1984, ISSN: 0734-2071. DOI: 10.1145/2080.357392. [Online]. Available: http://doi.acm.org/10.1145/2080.357392.

[8] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Distributed systems (2nd ed.)", in, S. Mullender, Ed., New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, ch. The Primary-backup Approach, pp. 199–216, ISBN: 0-201-62427-3. [Online]. Available: http://dl.acm.org/citation.cfm?id=302430.302438.

[9] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-15260-3.

[10] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems", *J. ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996, ISSN: 0004-5411. DOI: 10.1145/226643.226647. [Online]. Available: http://doi.acm.org/10.1145/226643.226647.

[11] B. Charron-Bost and A. Schiper, "Uniform consensus is harder than consensus", *J. Algorithms*, vol. 51, no. 1, pp. 15–37, Apr. 2004, ISSN: 0196-6774. DOI: 10.1016/j.jalgor.2003.11.001. [Online]. Available: http://dx.doi.org/10.1016/j.jalgor.2003.11.001.

[12] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study", *ACM Computing Surveys (CSUR)*, vol. 33, no. 4, pp. 427–469, 2001.

[13] G. Chockler, S. Gilbert, V. Gramoli, P. M. Musial, and A. A. Shvartsman, "Reconfigurable distributed storage for dynamic networks", *Journal of Parallel and Distributed Computing*, vol. 69, no. 1, pp. 100–116, 2009, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2008.07.007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731508001317.

[14] 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, Monterey, California: ACM, 2002, pp. 78–87, ISBN: 1-58113-485-1. DOI: 10.1145/571825.571837. [Online]. Available: http://doi.acm.org/10.1145/571825.571837.

[15] F. Cristian and C. Fetzer, "The timed asynchronous distributed system model", *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 642–657, Jun. 1999, ISSN: 1045-9219. DOI: 10.1109/71.774912.

[16] Diego, Ongaro, The Raft Consensus Algorithm: Implementations, https://raft.github.io/#implementations.

[17] E. W. Dijkstra, *Selected Writings on Computing: A personal Perspective*, 1st ed.: Springer-Verlag New York, 1982, ch. On the Role of Scientific Thought, pp. 60–66, ISBN: 978-1-4612-5695-3. [Online]. Available: https://doi.org/10.1007/978-1-4612-5695-3.

[18] P. Dutta, R. Guerraoui, R. R. Levy, and M. Vukolić, "Fast access to distributed atomic memory", *SIAM J. Comput.*, vol. 39, no. 8, pp. 3752–3783, Dec. 2010, ISSN: 0097-5397. DOI: 10.1137/090757010. [Online]. Available: http://dx.doi.org/10.1137/090757010.

[19] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process", *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985, ISSN: 0004-5411. DOI: 10.1145/3149.214121. [Online]. Available: http://doi.acm.org/10.1145/3149.214121.

[20] E. Gafni and L. Lamport, "Disk paxos", *Distrib. Comput.*, vol. 16, no. 1, pp. 1–20, Feb. 2003, ISSN: 0178-2770. DOI: 10.1007/s00446-002-0070-8. [Online]. Available: http://dx.doi.org/10.1007/s00446-002-0070-8.

[21] S. Gilbert, N. Lynch, and A. Shvartsman, "Rambo ii: Rapidly reconfigurable atomic memory for dynamic networks", in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, Jun. 2003, pp. 259–268. DOI: 10.1109/DSN.2003.1209936.

[22] S. Gilbert, N. Lynch, and A. Shvartsman, "Rambo: A robust, reconfigurable atomic memory service for dynamic networks", *Distributed Computing*, vol. 23, no. 4, pp. 225–272, 2010.

[23] Golang protobuf proposal: protoc-gen-go: provide first-class support for plugins, https://github.com/golang/protobuf/issues/547, Visited: 09.05.18.

[24] Google Inc, GCP Machine Types, https://cloud.google.com/compute/docs/machine-types.

[25] Google Inc, GCP Network Tiers, https://cloud.google.com/network-tiers.

[26] Google Inc, GCP Regions & Zones, https://cloud.google.com/about/locations/?region=europe#region.

[27] Google Inc, Google Cloud Platform, https://cloud.google.com.

[28] Google Inc, Google Compute Engine, https://cloud.google.com/compute.

[29] Google Inc, gRPC, http://www.grpc.io/.

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

[31] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi, "Incremental consistency guarantees for replicated objects", in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, 2016, pp. 169–184, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/guerraoui.

[32] L. Jehl, T. E. Lea, and H. Meling, "Replacement: Decentralized failure handling for replicated state machines", in *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, Sep. 2015, pp. 156–165. DOI: `10.1109/SRDS.2015.29`.

[33] L. Jehl, "Reconfiguration for fault tolerant services in an asynchronous system", PhD thesis, Faculty of Science and Technology, University of Stavanger, Jun. 2016.

[34] L. Jehl and H. Meling, "The case for reconfiguration without consensus: Comparing algorithms for atomic storage", in *LIPIcs-Leibniz International Proceedings in Informatics*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, vol. 70, 2017.

[35] L. Jehl, R. Vitenberg, and H. Meling, "Smartmerge: A new approach to reconfiguration for atomic storage", in *International Symposium on Distributed Computing*, Springer, 2015, pp. 154–169.

[36] L. Lamport, "Fast paxos", *Distributed Computing*, vol. 19, no. 2, pp. 79–103, Oct. 2006. [Online]. Available: `https://www.microsoft.com/en-us/research/publication/fast-paxos/`.

[37] L. Lamport, "On interprocess communication–part ii: Algorithms", *Distributed computing*, vol. 1, no. 2, pp. 86–101, 1986.

[38] L. Lamport, "Paxos made simple", *ACM SIGACT News (Distributed Computing Column)*, vol. 32, no. 4 (Whole Number 121), pp. 51–58, Dec. 2001. [Online]. Available: `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`.

[39] L. Lamport, "The part-time parliament", *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998, ISSN: 0734-2071. DOI: `10.1145/279227.279229`. [Online]. Available: `http://doi.acm.org/10.1145/279227.279229`.

[40] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication", in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, ser. PODC '09, Calgary, AB, Canada: ACM, 2009,

pp. 312–313, ISBN: 978-1-60558-396-9. DOI: 10.1145/1582716.1582783. [Online]. Available: http://doi.acm.org/10.1145/1582716.1582783.

[41] B. W. Lampson, "How to build a highly available system using consensus", in *Proceedings of the 10th International Workshop on Distributed Algorithms*, ser. WDAG '96, London, UK, UK: Springer-Verlag, 1996, pp. 1–17, ISBN: 3-540-61769-8.

[42] B. W. Lampson, "The abcd's of paxos", in *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '01, Newport, Rhode Island, USA: ACM, 2001, pp. 13–, ISBN: 1-58113-383-9. DOI: 10.1145/383962.383969. [Online]. Available: http://doi.acm.org/10.1145/383962.383969.

[43] T. E. Lea, L. Jehl, and H. Meling, "Gorums: New abstractions for implementing quorum-based systems", 2016, Unpublished (Extended Version).

[44] 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)*, Jun. 2017, pp. 2380–2385. DOI: 10.1109/ICDCS.2017.166.

[45] T. E. Lea, L. Jehl, H. Meling, and S. M. Pedersen, Gorums (source code), https://github.com/relab/gorums.

[46] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish, "Improving availability in distributed systems with failure informers", in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13, Lombard, IL: USENIX Association, 2013, pp. 427–442. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482667.

[47] H. Meling and L. Jehl, "Tutorial summary: Paxos explained from scratch", in *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304*, ser. OPODIS 2013, Nice, France: Springer-Verlag, 2013, pp. 1–10, ISBN: 978-3-319-03849-0. DOI: 10.1007/978-3-319-03850-6_1. [Online]. Available: https://doi.org/10.1007/978-3-319-03850-6_1.

[48] H. Meling, A. Montresor, B. E. Helvik, and O. Babaoglu, "Jgroup/arm: A distributed object group platform with autonomous replication management", *Software: Practice and Experience*, vol. 38, no. 9, pp. 885–923, 2008.

[49] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments", in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: ACM, 2013, pp. 358–372, ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2517350. [Online]. Available: http://doi.acm.org/10.1145/2517349.2517350.

[50] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", 2008.

[51] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm", in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14, Philadelphia, PA: USENIX Association, 2014, pp. 305–320, ISBN: 978-1-931971-10-2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643666.

[52] S. M. Pedersen, "A practical analysis of the gorums framework: A case study on replicated services with raft", Master's thesis, University of Stavanger, http://hdl.handle.net/11250/2455424, Jun. 2017.

[53] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields", *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[54] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design", *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984, ISSN: 0734-2071. DOI: 10.1145/357401.357402. [Online]. Available: http://doi.acm.org/10.1145/357401.357402.

[55] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial", *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990, ISSN: 0360-0300. DOI: 10.1145/98163.98167. [Online]. Available: http://doi.acm.org/10.1145/98163.98167.

[56] C. Shao, J. L. Welch, E. Pierce, and H. Lee, "Multiwriter consistency conditions for shared memory registers", *SIAM J. Comput.*, vol. 40, no. 1, pp. 28–62, Jan. 2011, ISSN: 0097-5397. DOI: 10.1137/07071158X. [Online]. Available: http://dx.doi.org/10.1137/07071158X.

[57] A. Shraer, J.-P. Martin, D. Malkhi, and I. Keidar, "Data-centric reconfiguration with network-attached disks", in *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, ser. LADIS '10, Zurich, Switzerland: ACM, 2010, pp. 22–26, ISBN: 978-1-4503-0406-1. DOI: 10.1145/1859184.1859191. [Online]. Available: http://doi.acm.org/10.1145/1859184.1859191.

[58] A. Spiegelman, I. Keidar, and D. Malkhi, "Dynamic reconfiguration: A tutorial (tutorial)", in *LIPIcs-Leibniz International Proceedings in Informatics*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, vol. 46, 2016.

[59] The Go Authors, The Go Programming Language, https://golang.org.

[60] M. Vukolic, *Quorum Systems: With Applications to Storage and Consensus.* Morgan & Claypool, 2012, ISBN: 9781608456840. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6813714.