University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science	Spring/ ~~Autumn~~ semester, 2020 Open / ~~Confidential~~
Author: Eivind Bakkevig	Eivind Bakkevig
Programme coordinator: Hein Meling Supervisor(s): Hein Meling and Thomas Stidsborg Sylvest	
Title of master's thesis: Implementing a Distributed Key-Value Store Using Corums	
Credits: 30 ECTS	
Keywords: Distributed Systems, Consensus Algorithms, Corums, Paxos, State Machine Replication	Number of pages: 60 + supplemental material/other: Code on Github Stavanger, June 30/2020 date/year

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

Implementing a Distributed Key-Value Store Using Corums

Master's Thesis in Computer Science
by

Eivind Bakkevig

Supervisors

Hein Meling

Thomas Stidsborg Sylvest

June 30, 2020

Abstract

Corums is a novel framework made for simplifying the process of building fault-tolerant systems. In this thesis, we investigate whether Corums is suitable for implementing a practical fault-tolerant service by using it to implement a distributed key-value store, which we call Distributed Dictionary. This service uses a Corums-based Multi-Paxos implementation to handle state replication. We assess the usability of Corums in terms of ease of adoption, availability and protection from user errors. We also run experiments to evaluate the performance efficiency of Corums in a system like ours. We discover that the abstractions Corums provides lead to more readable code and an execution flow that is easier to reason about than traditional programming paradigms. We also discover and discuss areas where the Corums framework should be improved.

Acknowledgements

I would like to thank my supervisor Hein Meling for his guidance and encouragement throughout the course of this thesis. I would also like to thank Thomas Stidsborg Sylvest for his technical help in learning to use Corums as well as his continuous improvements to the framework as issues were discovered.

Contents

Abstract i

Acknowledgements ii

1 Introduction 1
- 1.1 Project Description . . . 2
- 1.2 Contributions . . . 2
- 1.3 Outline . . . 3

2 Background 4
- 2.1 Distributed Systems . . . 4
  - 2.1.1 Why We Need Distributed Systems . . . 4
  - 2.1.2 Challenges of Distributed Systems . . . 7
- 2.2 Replicated State Machines . . . 7
- 2.3 Consensus Algorithms . . . 8
  - 2.3.1 The Paxos Protocol . . . 9
  - 2.3.2 Batching and Pipelining . . . 13

3 Corums 15
- 3.1 Motivation . . . 15
- 3.2 Network Communication Using Reactive Programming . . . 16
  - 3.2.1 Reactive Programming . . . 16
  - 3.2.2 Corums' Hybrid Programming Model . . . 17
  - 3.2.3 Corums Communication Streams . . . 18
  - 3.2.4 Corums In-Memory Network . . . 19
- 3.3 Single-Threading . . . 20
- 3.4 Persistency . . . 21

4 Implementing a Distributed Dictionary 22
- 4.1 Introducing Distributed Dictionary . . . 22
- 4.2 System Architecture for Distributed Dictionary With Paxos Using Corums 23
  - 4.2.1 Client Handling Module . . . 24
  - 4.2.2 Paxos Module . . . 26
  - 4.2.3 Failure Detection and Leader Election Module . . . 29
  - 4.2.4 Modifying the Corums Networking Implementation . . . 32

4.3 Development Environment 33
4.3.1 Using Corums In-Memory Network for Testing 33
4.3.2 Using Docker to Prepare for Benchmark Measurements 33

5 Corums Evaluation **35**
5.1 Corums Usability 36
5.2 Corums Compatibility 37
5.3 Distributed Dictionary Performance Efficiency 38
5.3.1 Experimental Setup 38
5.3.2 Experimental Results 39

6 Lessons Learned, Conclusion and Future Directions **43**
6.1 Lessons Learned 43
6.1.1 Learning a Framework With No Community 43
6.1.2 Lack of Documentation 44
6.1.3 Contributing to Corums 44
6.1.4 Learning C# 44
6.2 Future Work 44
6.3 Conclusion 46

List of Figures **46**

List of Tables **48**

A Experimental Data **50**

B Distributed Dictionary Source Code **51**

Bibliography **52**

Chapter 1

Introduction

Modern digital services are required to be fast, stable, durable and safe. An update on a global system performed by an individual in Southeast Asia needs to be visible to an another system user located in North America, instantly. Tens of thousands of users must be able to connect to a system at the same time, without experiencing a hiccup. When the power goes out in a data center due to an electrical fire, the user must remain oblivious. The engineering behind such achievements is highly complex and requires expert knowledge.

In this thesis, we look at tools that aim to assist in the process of developing such systems. We look at implementing Paxos [1–3], a renowned algorithm used to achieve consensus among machines in a distributed system. Paxos is used to achieve strong consistency in state machine replication in high-availability systems. It is a tried and tested algorithm that has proven it's worth over decades of development in the field of distributed systems.

We also look at Corums, a messaging framework that aims at alleviating software engineers from some of the implementation details of consensus algorithms, such as Paxos, which is known to be a complex algorithm to implement correctly. Corums is a new framework under development by the BBChain research group at UiS, headed by the supervisor of this thesis, Hein Meling.

The main purpose of this thesis is to implement a real-world application utilizing Corums to evaluate the gain in ease of implementation of consensus algorithms and to evaluate the performance of Corums in an application that depends on consensus algorithms. The result is a scalable and fault-tolerant replicated key-value store with an ASP.NET Core Web API [4] wrapped around to handle the communication with clients. This approach means that *any* web client can utilize this key-value store, which leaves all options for

tools used to test similar systems open to test our work as well, which we consider an important point for further development and use of the work done in this thesis.

1.1 Project Description

The project description given prior to starting our work was the following:

> The goal of this project is to implement a distributed highly available key value store with strong consistency guarantees using the Corums framework. Such systems are built using a consensus algorithm, e.g. Raft or Multi-Paxos at their core. The student is free to choose an algorithm in cooperation with the supervisors.
>
> Corums is an artifact of the current research, into simplifying implementation of consensus algorithms, conducted by the research group headed by Hein Meling. Corums is a .NET framework specifically designed for this purpose. This means that the key value store must be implemented in C#.
>
> The project provides the opportunity for the student to dig into and understand how enterprise systems such as Cassandra and distributed databases functions at a low level. Thereby, providing the student with invaluable knowledge before pursuing be it a professional or academic career afterwards. Furthermore, the project is very much open for the ambitious student.

1.2 Contributions

Taking the given project description into consideration, we decided on implementing Paxos to achieve strong consistency between multiple machines. High availability is achieved by having multiple machines and using them so that the system can keep functioning in the event of one or more server failures.

As the main focus of Corums is to simplify the implementation of consensus algorithms, the greatest contribution of our work was to come in with knowledge of consensus algorithms, but no knowledge of Corums, and implement a consensus algorithm using Corums. This way, we were able to provide new, valuable feedback to the developers of Corums, helping them discover framework weaknesses in terms of elements or abstractions that are hard to understand for its users.

In addition, we contributed by evaluating the performance of Corums as the backbone for communication in a realistic distributed environment. We stress-tested our service to measure the performance of our use of Corums under heavy load.

Finally, we contributed to the networking implementation of Corums by implementing our own TCP socket communication and integrating it with Corums.

1.3 Outline

The remainder of this thesis will have the following structure:

Chapter 2 introduces the reader to the relevant background info for the work done in this thesis. It discusses distributed systems and the challenges they come with. It introduces state machine replication and consensus algorithms. Finally, it explains the Paxos algorithm and some common optimizations to it.

Chapter 3 presents Corums. It will explain what the motivation behind Corums is and what Corums brings to the table when it comes to implementing consensus algorithms.

Chapter 4 will give a detailed explanation of our fault-tolerant distributed key-value store, which we have named Distributed Dictionary. It will give a general overview of how the system is structured, as well as going into how the important components are implemented.

Chapter 5 contains an evaluation of Corums as a software framework, as well as experimental results from running load-tests on our replicated key-value store.

Chapter 6 lists some of the lessons we have learned during the work on this thesis. It also suggests directions for future work on Corums and our service. Finally, it concludes the work we have done.

Chapter 2

Background

This chapter will explain the background of why this thesis work has been done. The information given in this chapter is considered a prerequisite to understand the work done in the practical solution and experiments. It will explain the general challenges that leads to the development of distributed systems, the concept of a replicated state machine as well as what a consensus algorithm is, why we need it and the way our choice of algorithm, Paxos, works.

2.1 Distributed Systems

A distributed system is a system running on multiple physical machines, passing messages over a network to coordinate actions to achieve a common goal. There are multiple approaches of distributed systems, some that aim to minimize the risk of data loss or downtime by having redundancy over multiple machines and some that aim to achieve higher performance by spreading the network traffic to multiple machines, either full-time or on-demand when the system is under heavy load. Some simply aim to improve productivity in system maintenance by splitting a system into components that each do their part in achieving a greater goal, which is typically referred to as a microservice architecture. Others are naturally distributed to be able to serve users or other systems in different geographical areas of the world.

2.1.1 Why We Need Distributed Systems

Why do we need to make systems more complex to implement by spreading their pieces to different physical locations? This introduces a multitude of new headaches that we

never would have to worry about if we just kept every system monolithic (running on one machine).

Availability

One of the most important factors used to evaluate a computer system's performance is *availability*. According to [5], availability in a computer system is defined as follows:

> "Availability means that a system is on-line and ready for access. A variety of factors can take a system off-line, ranging from planned downtime for maintenance to catastrophic failure. The goals of high availability solutions are to minimize this downtime and/or to minimize the time needed to recover from an outage."

If a power outage occurs in the building where a monolithic system is running, the service is gone. All active users will lose their connections, data in transit may be lost and important events will be missed.

If we consider a system that is distributed to machines in different physical locations with a reroute to a backup server in the case of an outage, this problem is solved instantly and thus, the availability of the system is dramatically improved. A system like this is referred to as a fault-tolerant system. Additionally, if the system's workload is spread to multiple components working on multiple machines (known as a microservice architecture), we can have partial failures, only rendering parts of the system unusable, while other parts will be working fine. This is not perfect, but obviously better than a full failure of the entire service.

A similar approach can be implemented in the case of planned downtime for maintenance. Instead of taking an entire service down to deploy a new version of the code base, a distributed system with the same processes running on multiple machines would allow the administrators to deploy to one machine at a time, while the rest of the machines are answering requests and performing their work as usual. Again, the distributed approach has dramatically improved availability.

Scalability

Another important point to be made for taking the distributed approach is a system's *scalability*. TechTerms [6] defines a scalable system as follows:

> "Scalable hardware or software can expand to support increasing workloads. This capability allows computer equipment and software programs to grow over time, rather than needing to be replaced."

Allowing a system to grow over time can mean multiple things. The size of a system can be referring to its number of users, the number of tasks it performs, the amount of data it processes etc. In this context, the meaning of a system's growth will be the amount of work it does, which all of the factors above affects.

A computer can only have so many resources, so if the amount of work necessary to run a service is increasing, the computer will at some point run out of these resources and be unable to handle more work. Taking the monolithic approach would mean the computer hardware would need to be replaced to something more powerful. This gets problematic in terms of financial cost when the workload gets large, as powerful hardware components are expensive. It's also limited how powerful a single machine can get. Taking the distributed approach solves this problem by having the work done by multiple machines, so that if the amount of work increases substantially, another machine can be added to the system to increase the total amount of available computing resources. If the workload is fairly stable, this can be a permanent upgrade. If the increased workload is only a temporary spike, a scalable system will also allow us to increase/decrease the amount of resources for limited periods of time.

Speed

An important metric to measure how well a piece of software is performing is to measure its speed in terms of how long it takes to perform any given task. This is also referred to as the latency between making a request to perform some operation and getting the response to that request.

If a monolithic system is slow and we want to do something to improve its speed, it could be a good solution to divide the work it's performing between multiple machines, depending on the nature of the operations it's performing. If the operations can be split into multiple independent tasks that each take some time to perform, chances are high that we could benefit from passing them to multiple "workers" that each execute their given task and return to the "master", who combines the results of the tasks to the result of the whole operation. For this to have a positive impact, it's important that the network latency of sending the messages between the "master" and "workers" doesn't outweigh the time we saved by not having the "master" do the work itself.

2.1.2 Challenges of Distributed Systems

What makes distributed systems so hard to implement?

- Unreliable communication
  When messages must travel over a network to reach a process running on a different machine, we run the risk of losing messages, losing connections to peers, receiving faulty messages due to signal noise, etc.
- No synchronized clocks
  One of the fundamental difficulties of ordering operations that are to be executed on multiple machines is that we can't trust that all machines have the same clock function. We cannot trust timestamps to decide ordering of messages.
- Concurrency/Parallelism
  Multiple processes executing at the same time may be using and updating the same resources. This requires synchronizing between the different processes.
- Slow communication
  Comparing distributed systems to monolithic systems, communicating between different parts/processes of the system is slow over a network compared to when all processes run on the same machine.

2.2 Replicated State Machines

The system we have implemented in this thesis is of a type that is commonly referred to as a finite state machine or simply a state machine. A state machine is an abstraction, meaning it's not a name of a physical concrete system, rather it's an abstract model we use as a generic way of speaking or writing about some computer systems, typically used when developing tools to be used for building such systems.

A state machine is any device that holds a state of something at a given time, as a result of a sequence of operations given from external inputs. The state must be deterministic based on this sequence of operations, meaning that if we have one state machine and would like to create another with the same state, we would execute the sequence of operations that have ran on the first machine and we should have the exact same state on the second machine as on the first machine.

A replicated state machine is when we have multiple instances of the same state machine that all have the same state and have had the same sequence of operations executed

on them. Replicated state machines is the principle the systems typically adhere to when we are creating fault-tolerant services to improve availability. Achieving such a system is most commonly done by utilizing a consensus algorithm, which is explained in Section 2.3.

2.3 Consensus Algorithms

Definition of consensus algorithms from [7]:

> "A consensus algorithm is a process in computer science used to achieve agreement on a single data value among distributed systems. Consensus algorithms are designed to achieve reliability in a network involving multiple nodes. Solving the issue — known as the consensus problem — is important in distributed computing and multi-agent systems."

To achieve the functionality of replicated state machines explained in Section 2.2 in a real-world application, we must ensure that all machines get the same operations executed on them to be certain that they all have the same state at all times. This is the problem that consensus algorithms solve.

As we have already discussed, when designing and implementing a distributed system, we operate with a mindset based on the infamous Murphy's law:

> "Anything that can go wrong, will go wrong."

Network partitions happen, power outages happen, computers crash for numerous reasons. Instead of doing everything possible to prevent these failures and design the system with an assumption of nothing going wrong, we use consensus algorithms in replicated state machines to ensure that the service stays up in the event of such failures.

In consensus algorithms, we use the word "proposal" as the terminology for a new operation that is to be executed on the replicated state machines, if consensus is reached. We also use the word "decision" as the terminology for when a consensus is reached and it has been decided that the proposed operation should be executed on all machines. In general, we aim to achieve the following safety criteria when implementing consensus algorithms [2]:

- Only values that have been proposed may be decided on.
- Only a single value can be decided on.
- No machines believe that a value has been decided unless it actually has been.

2.3.1 The Paxos Protocol

The consensus algorithm we have decided to implement in this thesis is called the Paxos protocol [1]. Paxos is a flexible consensus algorithm that has been used for numerous applications over the years. It was first introduced in [1] in 1998 by Leslie Lamport. The name originates from the Greek island Paxos and a fictional legislative system that Lamport imagined was used on the island. In this legislative system, the participants would walk in and out of the parliament as they pleased, and the system should still function as normal. This was meant to represent a cluster of computers where we expect some computers to be unavailable at some times. The explanation given in this article turned out to be perceived as very confusing and because of that, Lamport published a new paper [2] where he explained the protocol in a simplified manner. This has received criticism for being too simplified, as it takes away most of the complexity we encounter when we want to implement the protocol to be used for practical purposes. Either way, this is the explanation we use as ground material to explain the algorithm in this thesis, along with [3].

The Paxos protocol aims to satisfy the criteria listed in Section 2.3 with the following assumptions [2]:

> "Agents operate at arbitrary speed, may fail by stopping, and may restart. Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted. Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted."

It promises to satisfy the criteria if a majority of the machines in a cluster are up and running. If a majority of the servers in a system running Paxos stop functioning, the Paxos protocol won't be able to reach consensus anymore.

Paxos is explained by introducing us to three different roles, which are called "proposer", "acceptor" and "learner". These three roles all have different tasks that contribute to reaching a consensus when a new operation is requested by an external input. Each role may be running on its own physical machine, but the most common approach is for each machine in a distributed system to run all three roles.

Before we begin to explain the roles and the execution of Paxos, we need to introduce a few terms:

- **Broadcast**: In the context of consensus algorithms, a broadcast is used as a term for sending a message to all nodes participating in the consensus, which commonly

would be called a multicast in the networking field. It doesn't necessarily mean to send a message to *all* nodes on the network, which is the common definition of a broadcast in the networking field.

- **Paxos instance**: A full execution of the Paxos protocol from receiving a request with a value from external input to having decided on that value.
- **Quorum**: The criteria that must be reached for a value to be decided. In Paxos, this criteria is having a majority of the total number of servers in the system agreeing to something.
- **Paxos round**: An increasing unique number that is used to mark a value to be decided on and whether a message is logically "newer" than another (without taking actual time into consideration).
- **Client**: Some external entity which is sending requests to set new values to the system.

Proposer

The proposers initiate the Paxos instance. They take input from the process which handles client requests and then proposes to set a new value. When the proposer has gotten permission from the acceptors, it sends a request to set the value from the client request.

Acceptor

The acceptors accept proposals made by the proposer. They dictate whether a new value can be set and they ensure that new values aren't chosen until a quorum is reached.

Learner

The learners learn new chosen values from the acceptors. Eventually, they will conclude that a chosen value is decided. When this happens, they notify the client request handling process that the value requested has been decided, meaning that Paxos has reached a consensus on it.

Execution Flow

There are 4 different message types used to explain the Paxos protocol:

- **Propose**: Contains the round number.
- **Promise**: Contains the round number and, if a value has been decided, the decided value along with the round number belonging to it.
- **Accept**: Contains the round number and the value to be decided.
- **Learn**: Contains the round number and the value to be decided.

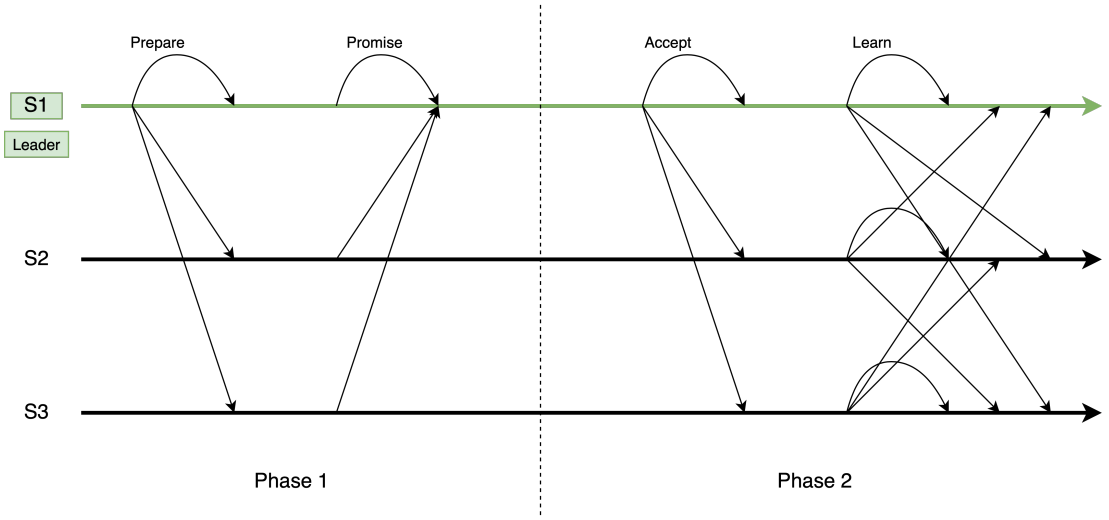

Figure 2.1: A perfect/ideal Paxos instance.

The execution of Paxos is often explained as two different phases, as illustrated in Figure 2.1.

Phase one starts when the system receives a client request. The proposer on the leader node initiates the Paxos instance by broadcasting a *propose* to all acceptors with a round number. This symbolizes that the proposer asks for permission from the acceptors to set a new value.

The acceptors who receive the propose checks if the round number contained within is higher than the highest round number they have seen before. If it is, they reply with a *promise* containing the round number, as well as the previously decided value with the highest round number associated, if any exist. The promise symbolizes that the acceptor promises not to accept any proposal with a lower round number than that which it includes in this promise. Additionally, the acceptors store the new round number from the received prepare as the new highest round number they've seen. Phase one is now finished.

Phase two starts when the proposer who initiated the Paxos instance has received a quorum of promises from the acceptors. If the proposer doesn't receive a quorum of promises in a given time, it will start phase one over again with a higher round number. After it has a quorum, the proposer broadcasts an *accept* to all acceptors, containing either the value with highest round number associated of the values in the received promises, if there were any, or the value from the client request. The accept also includes the round number it sent a propose for in phase one. This symbolizes that the proposer asks the acceptors to agree to setting a new value.

The acceptors who receive the accepts will check if the round number within is higher than any round number they have responded to before. If it is, they accept the value. If it isn't, they ignore/reject the value. If the acceptors accept the value, they finish their job by broadcasting a *learn* to all learners. This means that every learner should receive a learn from every acceptor.

Finally, all learners receive learns. When they have received a quorum of learns, all with same round number and value, the value in the learns is set on the machine they're running on. Phase two is now finished. On the leader node, a response to the client who made the initial request is sent.

Leader Election

For Paxos to guarantee *liveness*, meaning that something useful will eventually happen, we rely on what's known as a leader election algorithm [8]. The idea of this is to choose a distinguished machine by some criteria that will act as a leader and coordinator for the system. The leader will be handling the communication with external input sources and it will be the only machine that proposes new values. The reason this is necessary to guarantee liveness is that without it, we could easily end up in a scenario where multiple proposers try to execute phase one simultaneously and none of the proposals will ever be decided, because acceptors are seeing higher round numbers in *propose* messages constantly, meaning that all *accept* messages would be ignored, leading to an infinite loop of phase one attempts without making any progress.

There are many different approaches to failure detection and leader election, such as [9] and [10]. We won't go into details about how leader election algorithms function here, but it's important to know that Paxos relies on having one. The details of the leader election algorithm we implemented is given in Section 4.2.3.

Multi-Paxos

In regular Paxos, as described above, only one value can be decided. There has also been a lot of work done to be able to use Paxos for reaching consensus on multiple values by chaining instances together. This is known as Multi-Paxos and is one of the most popular uses of the Paxos protocol. There are many ways of implementing a chain of Paxos instances and Multi-Paxos simply refers to the concept of chaining instances, not a specific implementation, as there is no one implementation that will work for all applications. In Section 2.2, we talk about making sure all replicated state machines have the same sequence of operations applied to them. This sequence is known as a transaction log and Multi-Paxos is well suited to maintain such a sequence, as each operation is represented as a value to be decided by the Paxos protocol.

Implementing Multi-Paxos require some modifications to the Paxos algorithm as we have described it, typically involving a slot number to mark each decided instance with to ensure correct ordering. This also means that acceptors will have to inform proposers about *all* previously decided values, not just a single one.

Another important note about Multi-Paxos is that a very common optimization to implement for Multi-Paxos is to only require phase one on startup and leader changes. This means that as long as we have a stable leader, only phase two is run when new requests arrive. If the current leader crashes and a new leader is elected, the new leader must initiate and finish phase one before new client requests can be processed. This optimization reduces latency and increases throughput greatly in systems with a lot of traffic during short periods of time.

2.3.2 Batching and Pipelining

Request Batching

Further optimization to Multi-Paxos can be done by collecting requests from clients in a cache-storage without initiating a Paxos instance until we have reached a certain number of requests, called the batch size. This reduces the average latency, as storing a request and replying immediately is very fast compared to waiting for Paxos to reach a consensus. Once the batch size is reached, the leader will try to get a consensus for all the values at once. Obviously, this one large request will be slower than a single request is when running without batching, but as mentioned, the average latency and throughput will be improved overall as it takes less time to send one large proposal than *multiple* small ones.

Pipelining

Similarly to batching, pipelining is a technique used to reduce latency in Multi-Paxos under moderate to high traffic-load. Pipelining means that we allow a new Paxos instance to be started before the previous one has finished. This is especially beneficial in networks with high latency, while batching gives the largest performance gains in general [11].

More about the results of implementing batching and pipelining in Paxos is found in [11].

Chapter 3

Corums

Corums is a novel framework intended to simplify the development of consensus algorithms by providing useful abstractions that hide a lot of the complexity we normally would have to deal with. This chapter introduces Corums as well as explains some of the properties that make it stand out as a useful framework for programming consensus algorithms.

This chapter is inspired by [12], as well as good conversations with Thomas Stidsborg Sylvest.

3.1 Motivation

Implementing, operating and reasoning about consensus algorithms, such as Paxos, is notoriously difficult. The research being done by the developers of Corums has a focus on developing a software framework in a mainstream language that will aid in simplifying the development of consensus algorithms. The main focus of this framework is on source code readability and ease of adoption.

There have been attempts at making designated programming languages intended for implementing consensus algorithms, such as DistAlgo [13]. This language has abstractions that make reading the source code of consensus algorithm implementations as similar as possible to reading their pseudo-code. This makes for great readability, but it's not suitable for mainstream industry adoption, as professionals will lack the tools they are accustom to when developing in the mainstream programming languages.

Corums is implemented in C# and compiled for the .NET Core framework [14], which is a very popular open-source, cross-platform software framework used for a large variety of applications. This is important, as it sets Corums well up for mainstream industry adoption.

Other attempts have been made to implement frameworks for implementing consensus algorithms, similar to the approach that Corums is taking. An example of this is Gorums [15], which is a software framework implemented using the Go language. It provides high-level abstractions for sending/receiving messages to multiple nodes at the same time and the workflow is comparable to that of Corums. Gorums facilitates code readability similar to Corums, but it's concurrency model relies on the Go language's runtime scheduler, which makes it harder to debug and reason about implementations than the single-threaded nature of Corums, which we'll explain in Section 3.3. This point is mentioned briefly when explaining some of the difficulties of implementing Raft [16] with Gorums [15] in [17].

3.2 Network Communication Using Reactive Programming

3.2.1 Reactive Programming

The information given in this subsection about reactive programming is inspired by [18].

Corums uses a reactive programming model to abstract away the network communication. Reactive programming is centered around what's known as data streams and events. A data stream is an object that may be observed by others who are interested in the data on the stream. Data streams emit events when something has happened. The event can be for everything from receiving a new message to finishing an execution of a method to receiving input from a user, all depending on the application it's used in. The observers of the stream will react to new events by calling static event handler methods. Algorithm 3.1 is pseudo-code that illustrates how subscribing to a stream will typically look.

```
DataStream.onEvent(handleNewEvent);

handleNewEvent(event) {
    log("A new event was observed!");
}
```

Algorithm 3.1: Stream subscription example.

By using data streams we have access to a multitude of useful operators. The reactive programming model uses operators applied to streams that return new streams after the operator is applied. This allows the developer to chain operators in a very explicit way which makes it easy to see what the desired result is by reading the code. An example of stream operators is given in Algorithm 3.2. In this example, we apply the

"Where" operator to a stream of network replies, which will return a new stream of all the replies with a "success" field set to true. On the resulting stream, we apply the "Count" operator, which simply counts the elements in its input stream and emits a single event containing the number of elements it got.

```
var numberOfSuccessReplies = replies
            .Where(reply -> reply.success)
            .Count();
```

Algorithm 3.2: Stream operators example.

A data stream will typically handle multiple types of events, allowing the observers to choose which events they are interested in. In Algorithm 3.1, the observer subscribes to all events. Normally, different observers will be interested in different event types and each event type will have its own event handler method.

When programming in the reactive paradigm, one of the fundamental differences to programming in the object-oriented paradigm is that you are writing asynchronous code. The code you are writing happens when an event is emitted, independent to where the main flow of the program is currently working. Imagine an application with a graphical user interface (GUI) that retrieves information from some service on the internet. When the user clicks a button to fetch the information, the GUI must still respond to input from the user while waiting to receive data from the web service. A reactive programming model facilitates this behaviour by having the main program flow set an event handler on a stream subscription when sending the request and then moving on without waiting for an event.

3.2.2 Corums' Hybrid Programming Model

Corums utilizes a lot of the principals from reactive programming to make useful abstractions for consensus algorithm implementations, but it doesn't limit developers to only use the reactive programming model. Corums facilitates a combination of reactive and object-oriented programming, where the idea is that one can use the reactive model for what it's good at, then switch to the more traditional object-oriented model when it's needed/wanted.

```
var promises = await Bus
        .Broadcast(new Prepare(_round))
        .Collect()
        .Take(_majority)
        .Last;

if (promises.Any(promise -> promise.hasPreviousDecidedValue) {
    return;
}
```

Algorithm 3.3: Corums' combination of reactive and object-oriented programming.

Algorithm 3.3 is valid Corums code for initiating phase one in Paxos and it shows the flexibility of Corums' hybrid model as well as how the declarative code style lets a reader of the code to easily reason about what's going on.

3.2.3 Corums Communication Streams

As mentioned, Corums abstracts away the network layer by using streams and events. Sending a message to a node is as simple as putting an object on a stream. Corums also provides useful methods for broadcasting a message to all nodes, directly replying to a message we have received etc.

When configuring a Corums node, we provide the network details for all nodes in the cluster to Corums. When this is done, Corums opens a network connection to the nodes and sets up an adapter that reads messages from the network layer, deserializes them and places them onto the ***Input*** stream. The Input stream can be injected into classes that are registered to Corums. The classes can then subscribe to messages coming in with filters to get the message types they are interested in. They can assign callback methods that will execute when a new message is received. This concept is often referred to as static event handlers. Algorithm 3.4 shows an example of how the Input stream in Corums is injected and how a Paxos proposer would subscribe to the event of new *Promises* being received and assign an event handler to react to these events.

```
class Proposer {

    Input _input;
    Logger _logger;

    Proposer(Input input) {
        _input = input;
        _logger = new Logger("Proposer");
    }

    void init() {
        _input.WhereEnvelopeContains<Promise>()
            .ExecuteOnEvent(HandlePromise);
    }

    void HandlePromise(Envelope<Promise> promise) {
        _logger.Log("I received a new promise.");
    }
}
```

Algorithm 3.4: Corums network stream example.

When sending messages in Corums, we use a stream named ***Output***. At the same time that the adapter for placing incoming messages on the Input stream was set up, another adapter is initiated that subscribes to all events on the Output stream and when events occur, serializes them and sends them on the network connection. Using this abstraction is similar to that of the Input stream. It is injected in the same fashion as shown in Algorithm 3.4 and to send a message is done as shown in Algorithm 3.5.

```
_output.Emit(new Envelope(message, messageId, toNodeId, fromNodeId);
```

Algorithm 3.5: Sending a message with Corums.

In addition to the Input and Output streams for handling network communication, Corums provides a third convenience object named ***Bus***. The bus is an abstraction for all network traffic, both incoming and outgoing, and simply wraps around the Input and Output streams. It has a lot of convenience features, such as broadcasting a message to all nodes in the cluster and directly collecting replies to a sent message, as shown in Algorithm 3.3.

3.2.4 Corums In-Memory Network

In the early stages of developing a consensus algorithm implementation, we may not want to bother with the network layer and which physical machines we're running on

just yet. It's useful to be able to test our implementation rapidly without having to worry about deployments, networking issues etc. To solve this, Corums provides another useful tool called In-Memory Network. This allows us to simulate multiple participants in a consensus algorithm in one process running on one machine, where participants will communicate using the same network stream abstractions that is used for proper network communication. This can help reduce noise while implementing the behavior of the participants, and when the implementation is finished and we want to start testing on multiple physical machines, Corums can be configured to start using proper network connections and a single participant for each machine, all without having to make changes to the actual consensus algorithm implementation.

3.3 Single-Threading

Corums uses a built-in ***scheduler*** that handles scheduling of the execution of method calls. The method calls are kept on a queue. They are executed sequentially on a *single thread* by an event loop that dequeues one executable at time. A new method call is not executed until the previous has finished. The architecture of the scheduler and event loop is illustrated in Figure 3.1. This allows the users of Corums to not worry about synchronization between threads and makes it easier to verify correctness of implementations and debug issues, due to the lack of thread interleaving.

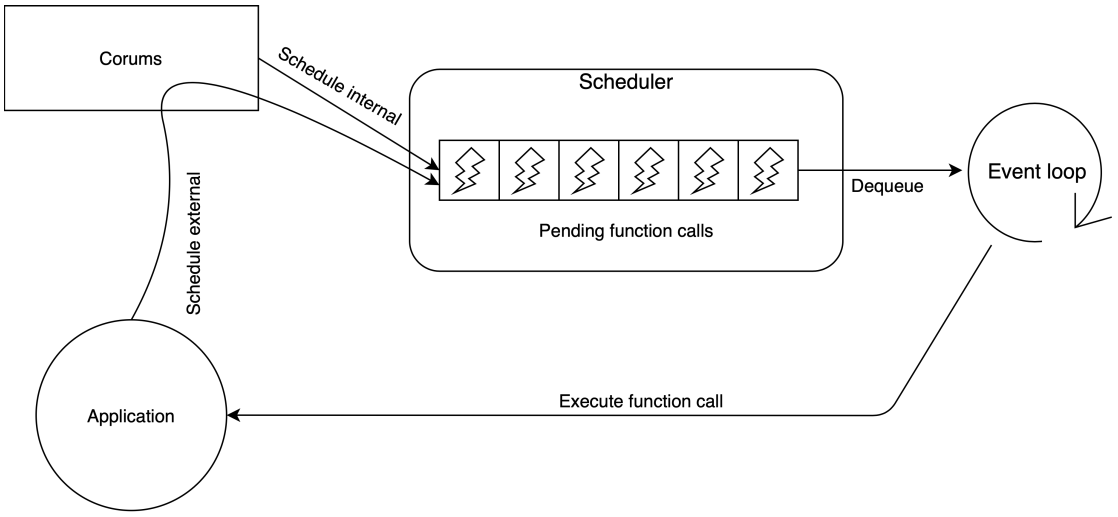

Figure 3.1: Corums' event loop architecture.

Both Corums internally and the user of Corums externally may schedule method calls. When an event is emitted on a stream that someone is subscribing to, the callback method will be scheduled for execution by Corums itself. If a code component outside of Corums gets a new client request to be delivered to a component residing inside Corums, the user may schedule a method call to handle the request, as shown in Figure 3.1.

3.4 Persistency

Corums has built-in functionality for automatic object state persistency. This means that Corums will automatically write the state of the relevant parts of the program to non-volatile memory as the state changes. Having this built-in persistency means that if a computer running a Corums program crashes in the midst of executing, it can be restarted and continue where it left off. The user of Corums controls which parts shall be serialized and written to disk, to then be deserialized and put into memory again upon a restart. The deserialization logic is invoked by the framework automatically on a restart and it will supply the latest serialized state.

Chapter 4

Implementing a Distributed Dictionary

This chapter will describe the path we have taken to solve the problem given in Section 1.1. It will describe the relationship between the different components we have used to achieve our result and how the resulting system functions.

4.1 Introducing Distributed Dictionary

As the main focus of this thesis is to explore the use of the Corums framework in practical applications that rely on consensus algorithms, we have built such a practical application with Multi-Paxos as the underlying mechanism to facilitate state machine replication. The name of the system we have built is inspired by the data structure for a key-value store in C# (and some other popular languages, such as Python), which is *"Dictionary"*, as well as the focus on each server in the cluster holding a copy of the dictionary, meaning it's *distributed*. The resulting name is Distributed Dictionary.

The purpose of Distributed Dictionary is to maintain a fault-tolerant key-value store by replicating the data across multiple physical machines. We want to achieve this by utilizing Multi-Paxos and we implement Multi-Paxos using Corums.

A key-value store must have three important operations available to clients:

- **Insert**: A request to insert a new entry in the key-value store. The client must provide a key and a value to be paired together as an entry.

- **Read**: A request to read an existing value from an entry in the key-value store. The client must provide a key that the server will use to retrieve the associated value.
- **Update**: A request to set a new value in an existing entry. The client must provide a key and a value. The provided key will be used by the server to look up the entry in which it will replace the existing value with the provided value.

Distributed Dictionary exposes these operations to clients in the form of endpoints in a Web API. By taking this approach, we ensure that any type of HTTP client, such as a web application, desktop program or mobile application, can use Distributed Dictionary.

Distributed Dictionary is designed for a single server, the leader, to communicate with the client. From the perspective of the client, there is only one server which holds the dictionary. To execute any of the mentioned operations, requests are sent to the leader, which coordinates all consensus logic, as well as communicating the results to the client.

4.2 System Architecture for Distributed Dictionary With Paxos Using Corums

In this section, we'll explain the system in detail. We'll explain the role of each of the important modules and how they are implemented. Figure 4.1 gives a technical overview of the architecture of Distributed Dictionary (including an external client). In this illustration we display three server replicas, but this is a configurable number and can be any number we choose when starting the system. It displays the client communicating over HTTP with all three replicas, which may seem strange, as we already have stated that the client only communicates with the leader. This is done to show that the client *can* communicate with all replicas, and if the current leader crashes, it will start to communicate with a new leader.

When starting Distributed Dictionary, we set a few environment variables in our deployment scripts. One containing a string of comma-separated hostnames that tell each replica which machines are participating. In addition, we give each replica a unique node identifier which is used internally by Corums. Finally, we also set the desired batch size (ref. Section 2.3.2) in an environment variable. This makes it easier to experiment with different configurations when testing the system, as it doesn't require a new version of the code deployed. Instead, to test a new configuration, we simply stop the program and start it back up with new values for the environment variables.

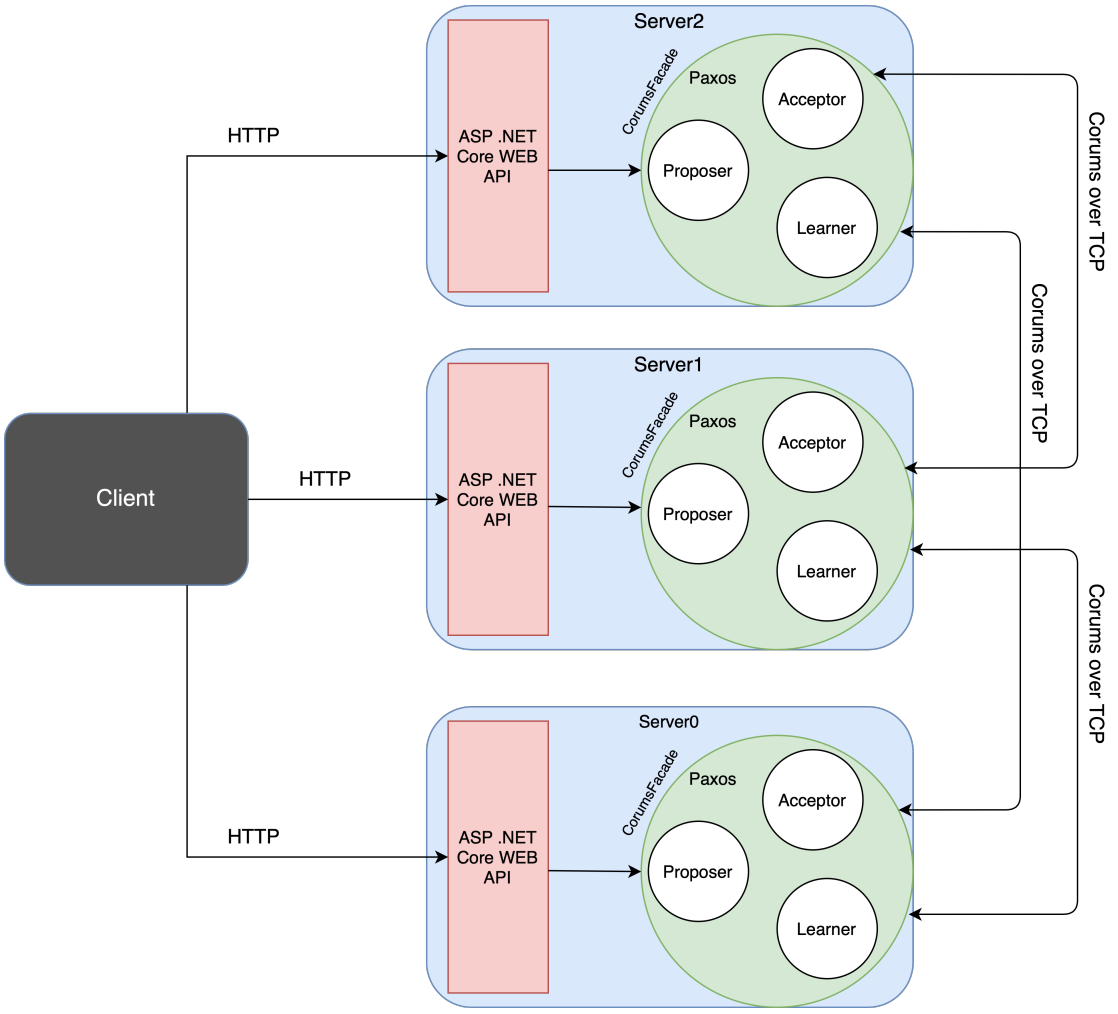

Figure 4.1: Distributed Dictionary system architecture.

4.2.1 Client Handling Module

The client handling module is the part of the program that takes HTTP requests from the outside, routes them to the correct place and sends a response to the client saying what the result of its request was. It's the entry point of the execution flow for any operation in Distributed Dictionary.

Referring to Figure 4.1, the client handling module is the red square marked with "ASP.NET Core Web API" in every server replica. This module consists of two main components: the controller and the request handler.

Controller

The controller is set up as an ApiController in the ASP.NET Core Web API framework [4]. Using ApiController gives us a lot of useful features automatically. It allows us to

easily route URLs to methods in the code, automatically deserialize JSON input to our data models and reply to clients with specific status codes and messages that explains the result of their requests.

```
[HttpGet("getValue")]
public ClientResponse GetValue(string key) {
  var value = _requestHandler.GetValueFor(key);
  return new ClientResponse(value.Value, $"The value for [{key}] is [{value}].");
}
```

Algorithm 4.1: Defining an API endpoint in an ApiController class.

Algorithm 4.1 shows how we set up the endpoint for the read operation described in Section 4.1. The first line defines an endpoint at the path /getValue which takes HTTP GET requests. The endpoint is linked to the following method, which takes a key as a string input, which is automatically parsed from the request URL. This key is passed along to the request handler to retrieve the associated value. Finally, a client response containing the value and a human-readable message is created and returned. The response object is automatically serialized to JSON and sent to the client. Because we haven't provided an HTTP response code, the code 200 will automatically be used, indicating a successful response.

In addition to the /getValue endpoint, we have two other endpoints defined in the same ApiController:

1. **GET - /getDictionary**: Takes no input. Returns the entire dictionary with all entries.
2. **POST - /insertOrUpdate**: Takes a key-value pair as input. Inserts a new entry if no entry is present for the provided key. If an entry is present for the provided key, updates that entry with the provided value.

Request Handler

The request handler is the next step in the flow of a client request passing through the system. As shown in Algorithm 4.1, a request to get a value associated with a provided key is passed from the controller to the request handler. The request handler will then ask the Paxos module for the value, before returning it to the controller.

The communication with the Paxos module from the outside is done through the CorumsFacade abstraction, illustrated as the green circle in Figure 4.1. We use the

CorumsFacade to schedule a new method call to be executed by Corums' scheduler, described in Section 3.3.

```
_paxosNode.Scheduler.Schedule<Coordinator, DictionaryValue>(coordinator =>
    coordinator.GetValueFor(key)).Result;
```

Algorithm 4.2: Calling a method through the CorumsFacade.

Algorithm 4.2 shows how we retrieve a value from the Paxos module through the CorumsFacade. The _paxosNode object is the CorumsFacade. Coordinator is the class that contains the method to get a dictionary value, which we'll explain in detail in Section 4.2.2. DictionaryValue is the return type of the method we're scheduling. Finally, calling .Result on the returned Task [19] means that we want to wait for the method to be executed by the scheduler so we can have our dictionary value before moving on.

In addition to handling the communication with the Paxos module, the request handler takes care of request batching as described in Section 2.3.2. This requires thread synchronization, as each request coming into the controller will be handled by a new thread. To be sure that we pass along a batch of requests when it is exactly the batch size we want, we must ensure that only one thread is allowed to add its incoming request to the request buffer at a time, followed by checking if the buffer size is equal to the batch size. We do this by utilizing the built-in locking feature in C# [20] to lock on a common object around the critical lines of code, making the next thread that wants to execute the same method have to wait for the lock to be released to execute these lines.

For each request that is added to the buffer before it's full, we simply return nothing (null) to the controller, which is interpreted as a signal that the request has been added to the buffer and the desired operation will be executed when the buffer is full. This results in a 202 client response with a corresponding message to the client. When a client request results in the batch size being reached, the batched operations will be delivered to the Paxos module and we will wait for a decision to be made before passing the decision to the controller, which replies to the client with a 200 and a corresponding message.

4.2.2 Paxos Module

The Paxos module is the main use of the Corums framework in this thesis. It implements the Multi-Paxos protocol as described in Section 2.3.1 running all three roles on every machine, with some minor adaptions to fit our use case. The three Paxos roles: Proposer, Acceptor and Learner are implemented as separate components interacting using the

stream abstractions presented in Section 3.2.3. Communicating between these roles on the same machines is not differentiated from communicating between these roles over the network to other machines.

We utilize the Bus to *broadcast* propose, accept and learn messages as well as *sending* a promise as a reply to a server who initiated Paxos' phase one. In addition, on program startup, we set up static event handlers to take action when messages of different types are received on the Input stream. The setup and the event handler method for handling prepare messages on the acceptor is shown in Algorithm 4.3. Registering the event handler is done in a separate method that is executed on startup, but for simplicity we chose to extract the relevant line for this purpose, rather than showing that whole method. The same approach for registering the event handler as shown in Algorithm 4.3 has been taken for all incoming messages on all three Paxos roles.

```
_input.WhereEnvelopeContains<Prepare>()
    .ExecuteOnEvent(true, HandlePrepare);

private void HandlePrepare(Envelope<Prepare> envelope) {
    var prepare = envelope.Message;
    _logger.Log($"Received prepare: [{prepare}] from [{envelope.From}].");

    if (prepare.Round > _round) {
        _round = prepare.Round;
        _bus.Send(new Promise(_round, GetSlotsHigherThan(prepare.Slot)),
            envelope.From);
    }
}
```

Algorithm 4.3: Setting up an event handler for receiving prepare messages on the acceptor.

Behavior On Startup

When the program starts, all the event handlers for incoming messages are set up on the proposer, acceptor and learner. In addition, every proposer will check if it's running on the leader node. The one proposer which finds out that it *is* running on the leader node, will wait five seconds to allow others to start up and then it will initiate Paxos phase one by broadcasting a prepare and waiting to receive promises from the acceptors. If it doesn't receive a majority of promises within a second, it increases its round number and tries again. This is repeated until a majority of promises is received or the proposer finds out it's no longer the leader. When phase one is finished, all Paxos roles will remain inactive until external input arrives.

Behavior When New Values Arrive

When a new value arrives from the client handling module, the proposer will start by checking if phase one is finished. If it isn't, the client request is put into a temporary storage and kept there until phase one finishes. If phase one *is* finished, the proposer will immediately broadcast an accept to the acceptors followed by the acceptors broadcasting a learn to the learners while also storing the value as a "filled slot".

As mentioned in Section 2.3.1, acceptors must inform proposers about all previously decided values that the proposer doesn't know about, so the reason for storing the accepted values is so that they can be included in a future promise in the event of receiving a prepare with a lower slot number than the acceptor has accepted before.

When the learners have received a majority of learns, a decision event is emitted. When a decision event is received, the requested operation(s) is performed on the actual dictionary. Finally, the leader node will notify the client handling module that consensus has been reached.

Coordinator

As we have briefly explained before, the interaction with the Paxos module from the outside must be done by scheduling method calls through the CorumsFacade. Because we have a client handling module *outside* of the CorumsFacade, which requires the ability to schedule a consensus round and wait for its result before replying to the client, we need a single place to both trigger an accept message to be sent from the proposer *and* listen for a decision event emitted from the learner. In other words, we need a single component inside the CorumsFacade that can coordinate the actions desired by the client handling module.

To facilitate this we created a fourth component in our Paxos module which we called the coordinator. The coordinator only coordinates consensus rounds on the machine its running on and does not send any messages over the network. Instead, it simply utilizes the Corums streams to emit internal events notifying the proposer about new client requests, as well as subscribing to events for decisions from the learner.

The coordinator also holds the actual dictionary. When decision events are received, it is the coordinator who performs the requested operation(s) on the dictionary. It's also the coordinator who provides the client handling module with the data used to reply to requests for reading a value from the dictionary or the entire dictionary. Because Corums has built-in persistency, the dictionary is simply kept as an in-memory field

variable in our implementation. When a replica is restarted, the dictionary will be read from persistent storage and put back into memory by Corums, providing the state it was in before restarting.

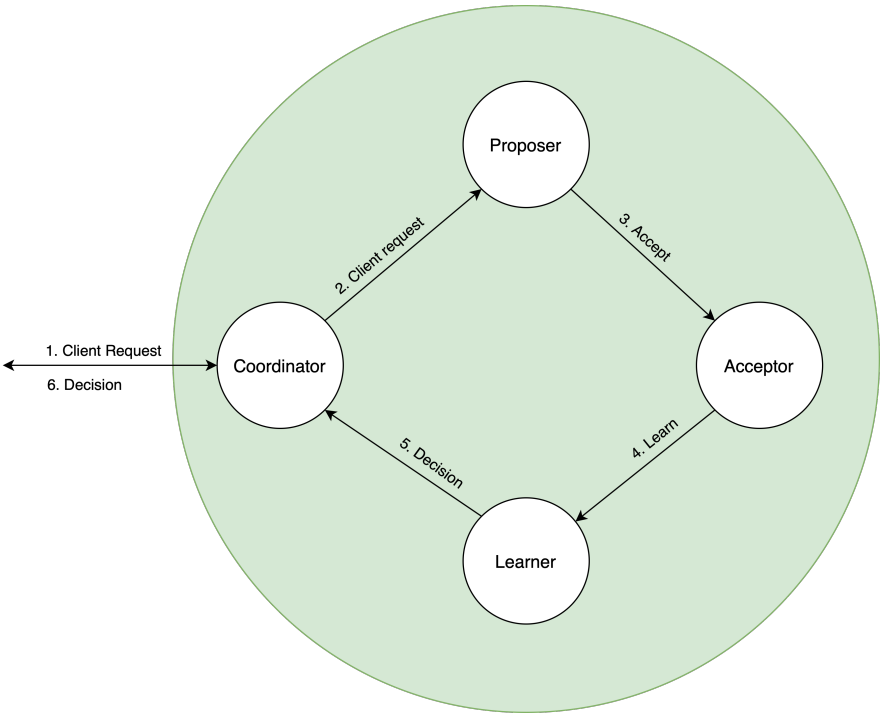

Figure 4.2: Overview of our Multi-Paxos implementation.

The flow of messages in Paxos phase two in our Paxos module using the coordinator is illustrated in Figure 4.2. A client request is passed from the client handling module to the coordinator by a simple method call, which eventually returns a decision when it is received from the learner. This method call is shown in Algorithm 4.2.

4.2.3 Failure Detection and Leader Election Module

In Section 2.3.1, we mentioned that for Paxos to guarantee liveness, we depend on having a leader election component available. To provide a leader election component, we also must implement a failure detector. We have implemented a failure detector and a leader elector combined in one component, loosely based on the Eventually Perfect Failure Detector, described in [21] and [22].

Failure Detection Implementation

When implementing our failure detector we were faced with two options:

1. Using a separate communication channel dedicated to this purpose.

2. Using Corums as the communication channel for this purpose in addition to the Paxos implementation.

The benefit of option 1 is that even if Corums for some reason struggles to process messages intended for other components, but the program hasn't fully crashed, the failure detector still survives and maintains correctness. On the other hand, if Corums doesn't function properly on a node, it has, as far as we're concerned, failed and won't be able to contribute anymore anyway. In our eyes, option 2 gives the failure detector a better understanding of the situation in the system, so we chose that approach.

Our failure detector contains two collections of node IDs: one that has the ID of all nodes that we consider alive and one that has the ID of all nodes that we suspect are crashed. To start with, no nodes are considered alive and no nodes are suspected. The execution in our failure detector mainly consists of two eternal-running loops: the heartbeats loop and the detection loop.

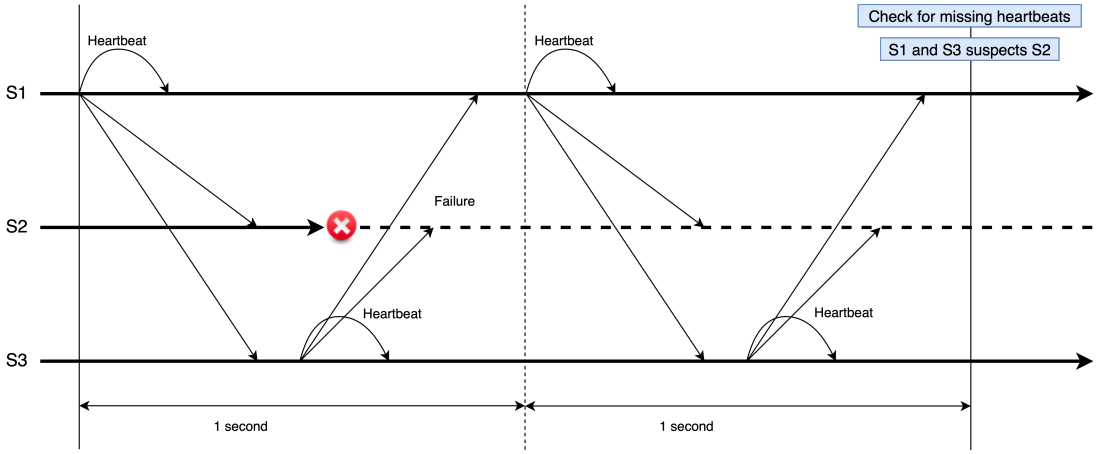

Figure 4.3: Failure detector execution flow.

The heartbeats loop simply utilize the Corums Bus to broadcast a heartbeat message to all other participants. A heartbeat message is an empty message only intended to tell others "I am alive". After broadcasting a heartbeat, the loop will sleep for one second, before running again. This is illustrated in Figure 4.3, where S1 and S3 run the heartbeats loop twice, while S2 crashes before broadcasting any heartbeats.

Whenever the failure detectors receives a heartbeat on the Corums Input stream, we add the senders node ID to the collection of alive nodes.

The detection loop has the job of checking the status of all participants. It starts by checking if the node's ID is present in any of the collections containing alive and suspected nodes. Based on the results of this, there are four possible outcomes:

1. The node is present in the alive collection, but not in the suspected collection. In this case, no action is taken, as this means the node is functioning as expected.

2. The node is not present in either of the collections. This means the node was alive on the last execution of the detection loop, but we haven't received a new heartbeat since then. In this case, we add the node to the suspected collection, as this means the node is not functioning as expected. This is what happens in Figure 4.3 where S1 and S3 suspects S2.

3. The node is present in the suspected collection, but not in the alive collection. In this case, no action is taken, as this means the node is still not functioning as expected, so it should remain in the suspected collection.

4. The node is present in both the suspected collection and the alive collection. This means the node has been suspected before, as it has failed to send a heartbeat in the given time, but has now started sending heartbeats again. In this case, we remove the node from the suspected collection, as this means the node is now functioning as expected again.

After one of the four outcomes has occurred for every participant, the detection loop empties the alive collection and then sleeps for two seconds to ensure that all participants have enough time to send at least one heartbeat before running the detection again.

Leader Election Implementation

The leader election algorithm we have implemented is defined in [21] and is called a monarchical leader election algorithm. In short, this means we assign a rank to each node, which will be used to decide who shall be leader. The leader will be the alive node with the highest rank. In our case, we have used the node ID, which has an integer value, to decide the rank. The higher the numerical value of the node ID, the higher the rank in the leader election algorithm.

In practice, we start by letting the proposer choose the node with the highest ID out of all participants as the leader. After all nodes have been checked for one of the aforementioned four outcomes in the detection loop, the leader election algorithm is asked to re-evaluate who shall be leader. It will go through all the nodes which are not suspected and, from these, select the node with the highest ID as leader. As the proposer is the component in our system that relies on knowing who is leader at all times, the leader election algorithm will compare the leader it has chosen with who the proposer believes is the leader. If the leader elector has another opinion than the proposer, it will notify the proposer about a

leader change. This will cause the proposer on the new leader node to initiate Paxos' phase one.

4.2.4 Modifying the Corums Networking Implementation

In Section 3.2.3, we explained how Corums has something we refer to as adapters, which coordinate the interaction between the network layer code and the Corums streams. Due to issues with instability and a desire to make some minor changes in the TCP communication in Corums, we ended up implementing our own TCP communication and thus, our own adapter logic as well. This was implemented as a separate module that we configured Corums to use instead of the built-in TCP communication when setting up our Paxos module.

The built-in implementation for TCP communication is quite complex, as it aims to guarantee delivery for all messages. It uses acknowledge messages to keep track of who has received a message and retries sending messages if it doesn't receive an acknowledgement. When we wanted to make changes, we found the complexity a hurdle, so we decided to simplify the logic to something more minimal and easier to debug and make changes to. We copied parts of the built-in logic, but left out the acknowledgements and retrying logic to end up with a minimalist implementation that consists of three components: one for receiving messages, one for sending messages and one that holds the network hostname of all the participants.

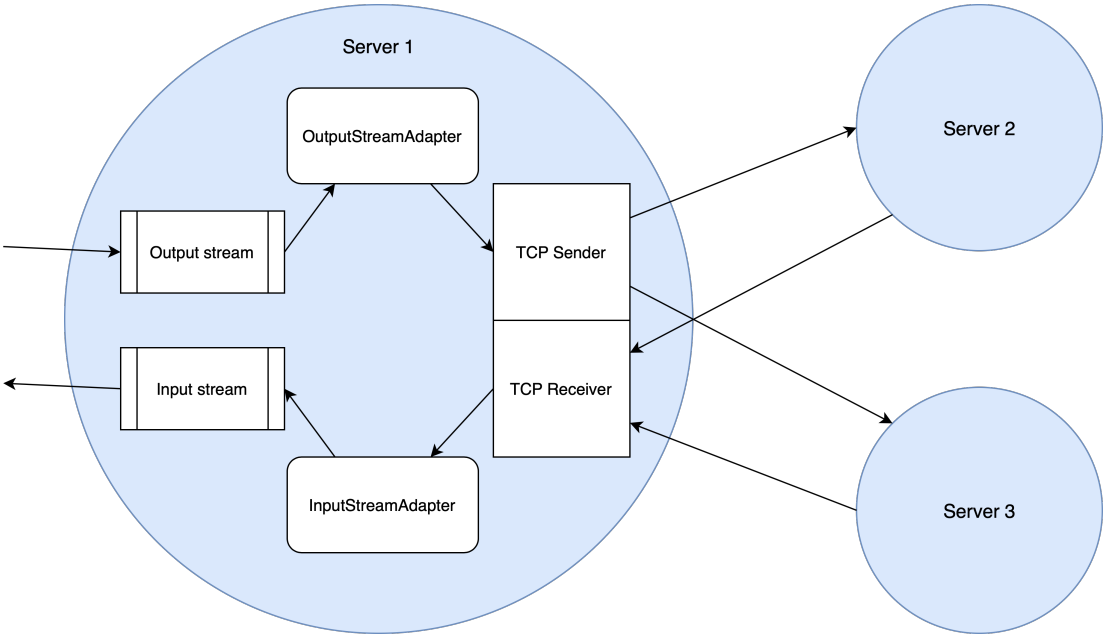

Figure 4.4: The structure of the Corums networking implementation we implemented.

The receiving component listens for incoming connections from other participants on a dedicated network port, then receives new messages, deserializes them and passes them onto the Corums adapter.

The sending component opens and maintains a TCP connection to all other participants, and when it receives a new message to be sent from the Corums adapter, it serializes it and sends it to the correct recipient.

The two Corums network adapters we needed to implement were the InputStreamAdapter and the OutputStreamAdapter.

The InputStreamAdapter has the job of passing messages received on the TCP socket to Corums. It has a method which is called by the TCP receiver component every time a new message is received, that simply emits a new event on the Input stream containing the received message.

The OutputStreamAdapter has the job of listening for new messages to be sent and passing them onto the TCP sending component. To achieve this, it subscribes to new events on the Output stream and, when an event occurs, it extracts the message and passes it onto the TCP sending component.

4.3 Development Environment

4.3.1 Using Corums In-Memory Network for Testing

As we were developing our Paxos implementation, we set up a convenient environment that allowed us to easily test our changes using the Corums in-memory network. When starting our program using our IDE, Corums is automatically configured to use the in-memory network and three virtual Paxos nodes would automatically be set up, all running in the same process. The virtual node with the highest ID would be leader and would handle the requests coming in on the HTTP endpoints, mocking the behaviour we would have when running on multiple machines. This setup allowed for testing changes in our algorithms as easy as pressing a single button in our IDE.

4.3.2 Using Docker to Prepare for Benchmark Measurements

When we got closer to having a system that was ready to be deployed to a proper cluster for making our benchmark measurements, we utilized Docker [23] to test in an environment as close to the real cluster as possible, but still maintaining the rapid nature

of testing locally. Docker is also useful for testing our networking implementation as, though it's running locally on our development machine, it uses the network interface card on the machine and a simulated network with DNS look-up etc.

Setting up a Docker environment requires us to build a Docker image for our program, that is used to set up containers. To create a Docker image we wrote a Dockerfile which downloads the .NET Core SDK base image [24] and uses that as a platform to build our application on, then downloads the ASP.NET Core runtime image [25] and uses that as a platform to run the compiled code on.

Once we have a Docker image ready, we use a tool called Docker Compose [26] to easily run our images on multiple containers. Docker Compose uses a YAML file to define our cluster in terms of the number of containers we're running, the hostname and IP address each of them will have, etc. When we have a setup we're happy with, starting the cluster of containers running Distributed Dictionary is as simple as writing "docker-compose up" in a terminal shell.

The Docker setup we have made is very convenient when developing consensus algorithms using Corums, and we encourage future projects involving Corums to utilize this contribution as well.

Chapter 5

Corums Evaluation

This chapter will explain how we evaluate the system we have built, described in Chapter 4. It will explain the criteria we used to assess Corums and the performance of our replicated key-value store using Corums. Finally, it will present the results we got on the criteria we researched.

To assess whether Corums provides value as a framework for working with consensus algorithms, we must decide on what criteria we measure its value. The ISO standard 25010:2011 [27] provides a framework for measuring software quality. It's designed for evaluating software used by industry or consumers, not necessarily for evaluating software tools to be used for creating other software, such as Corums. Nonetheless, some of it is suitable for Corums, so we have chosen three of the criteria from the product quality model in ISO 25010:2011 that we believe to be important qualities of a framework. These criteria are:

- **Usability**: How appropriate is this software for this use case? Is its features recognisable for people using it? How hard is it to learn? How well does it protect against user errors? How accessible is it?
- **Compatibility**: To what degree can this software co-exist with other tools? How does it interact with other tools?
- **Performance Efficiency**: How fast is the software? How does it utilize the machine's resources?

5.1 Corums Usability

As the main focus of Corums is simplifying consensus algorithm implementation by providing intuitive, useful abstractions, researching whether it does this should be a large part of assessing its worth. The problem with this is that how intuitive and useful something is is highly subjective by nature and therefore hard to measure. Our approach is to give our opinion on the experience we've had using Corums, as well as comparing the code in our Paxos implementation with another implementation that doesn't use Corums.

An important point to assess for how usable a software framework is, is how readable the resulting code using the framework is. This is crucial because for a software to have a long life, it must be easy to maintain over time and doing so generally includes multiple developers working on the same code base, which makes them reliant on being able to read each others code. A study [28] has been made where research is done to investigate what effect using the reactive programming paradigm has on program comprehension. They do this by implementing a program in two versions: one using the reactive paradigm and another using the traditional object-oriented style, then they conduct a test on thirty-eight subjects. The conclusion of this study is that understanding the reactive programming version requires less programming skills as that code is more readable.

Our experience with Corums coincides with the conclusion of the research on the effect reactive programming has on code readability. The abstractions provided and the flexibility in Corums of alternating between the reactive paradigm and the object-oriented paradigm, as described in Section 3.2.2, frees us from the bounds of any of the paradigms and allows us to focus on efficiency and readability. Our experience is that using Corums allows us to write code that is more similar to the pseudo-code we see in research articles for consensus algorithms, if we so desire. This is good because it makes it easier to reason about the correctness of implementations, as well as increasing the probability that those who are familiar with the protocol will be able to read and understand our code. Moreover, thread programming is a complex field, so, because Corums alleviates its users from that complexity, it protects them from making user errors.

We studied a Paxos implementation in Java called JPaxos [29, 30] to compare its code with ours. It could be considered unfair to compare implementations using two different languages, but Java is a language with a very similar syntax to C# [31, 32], as both are based on C and C++. In this study we found that the JPaxos code included a lot more configuration for setting the network connections up, routing incoming messages to the correct handlers, thread synchronization, etc. In those regards, Corums takes away the need for almost all of this configuration by handling it for us. Moreover, we discovered

that the message handlers in the Paxos roles are longer and harder to read than ours. As an example, the method for handling a prepare message in JPaxos is thirty lines long, while the equivalent method in the Distributed Dictionary Paxos implementation is nine lines long. Using lines of code as a metric for comparing algorithm implementations is somewhat unfair, as JPaxos handles a lot more edge cases than we do, but it *is* a reoccurring theme when comparing the two code bases that JPaxos has more code lines that are not directly a part of the Paxos protocol. In addition, we discovered that the abstractions Corums provides us encourages us to put thought into naming variables, methods, classes, etc. As a result, the code can be read more like reading the English language than the JPaxos code, resulting in greater program comprehension. The JPaxos implementation has also been split more granular into separate methods than the Distributed Dictionary Paxos implementation, making the execution flow harder to follow.

Additionally, we have first-hand experience with implementing Multi-Paxos using the Go programming language [33]. Go has some useful data structures and abstractions that help in protecting users from making errors in concurrent programming, such as goroutines and channels [34]. That being said, we still spent a large portion of our efforts in implementing Multi-Paxos with Go on setting up goroutines to listen for incoming messages, encoding/decoding message types, setting up network connections to peers, etc. Spending so much time on these things feels like noise that draws our attention away from the problem we're suppose to be solving. When comparing the experience we had with Go with the experience we've had in this thesis work with Corums and C#, it's clear that Corums simplifies the process greatly.

5.2 Corums Compatibility

Corums is distributed as a Nuget package [35] that can be included in any .NET Core project using the Nuget package manager. This is the standard mechanism of sharing code in the .NET world, and thus, it makes Corums very accessible for anyone to use.

The Stack Overflow Developer Survey for 2020 [36] shows that the ASP.NET Core framework is the sixth most used web framework with 19.1% of the participants answering that they use it. Moreover, in the category of other frameworks, libraries and tools, .NET Core ranks as the third most used with 26.7% answering that they use it. In addition, the survey presents the most loved *web* frameworks and most loved other frameworks, libraries and tools. ASP.NET Core takes the number one position as the most loved web framework, while .NET Core takes the number one position as the most loved other framework. This tells us that the .NET Core community has a strong position in the

industry and because it's loved by so many, we consider it likely to have a growing position in industry popularity in the coming years.

Because Corums is compiled to be used in the .NET Core runtime and can easily be included in any .NET Core project, it will be a compatible tool for a large variety of development projects.

As we demonstrated in Algorithm 4.2, Corums is well adapted for coordinating other application logic with Corums logic. Because consensus algorithms are usually just a minor part of practical applications, it's important that using Corums doesn't hinder any other features of the application. We have demonstrated in our system, which uses the ASP.NET Core Web API framework for the application logic, that Corums can be used in a practical application without causing problems for the rest of the program.

5.3 Distributed Dictionary Performance Efficiency

5.3.1 Experimental Setup

To measure the performance efficiency of Distributed Dictionary, we ran experiments to measure latency at different throughput settings on different configurations. All benchmarking experiments are run on the BBChain research group's cluster. We have used five server replicas running on five different physical machines in a LAN environment, each having an eight-core CPU and 32 GB of RAM. In addition, we have a single client running on a separate machine which simulates multiple clients by using thread concurrency. All machines run Ubuntu 18.04.4 as the operating system and the Distributed Dictionary server code is compiled using .NET Core 3.1.

The replicated service we run in the experiments is the Distributed Dictionary presented in Section 4.1. For the benchmarks, we request a high number of *update* operations on the key-value store. We run the experiment for two different batch sizes and we use pipelining in both cases. Refer to Section 2.3.2 for the explanation of these principles.

To set up Distributed Dictionary on the BBChain cluster, we wrote shell scripts that allowed us to compile the code, copy the code to all five machines and start the program on all machines by executing a single line in the terminal shell on our development machine. The shell script takes the desired batch size as input, so we can easily configure different batch sizes when running our experiments. We wrote extensive logging in our Paxos implementation, that we had printed to the terminal shell in which we executed the script. This allows us to follow what's going on on the servers while the automated client requests are executing.

JMeter

On the machine used for making the client requests, we utilize a tool from Apache called JMeter [37]. This is a cross-platform, open-source Java application made primarily for load-testing web applications, but it has also been expanded to support other test functions. This tool allows us to configure how many requests an experiment should consist of, how many simulated clients should be used, the rate to send the requests at, the connect-timeout, the response-timeout, etc. Moreover, when a test is running, JMeter reports the experienced latency along the way and when it finishes, it prints a summary of the test, which we noted down for each run to create the plots in Section 5.3.2. The data can be found in Appendix A.

5.3.2 Experimental Results

In this experiment we assess the performance of our Multi-Paxos implementation in Distributed Dictionary by sending a constant stream of update requests to the system and measuring the latency for a reply. As mentioned, we use batching and pipelining for all tests and we run the experiment for two different batch sizes. The batch sizes we have used are 64 and 1024. For each test we have made one million update requests to the leader from 1250 simulated clients. All the latency measurements are results of running three tests for each throughput setting and calculating the average of the latency result we get from JMeter, which again is the average latency for all executed requests in that test.

Batch Size: 64

Figure 5.1 shows the results of the experiment with a batch size of 64. The lowest throughput settings produces an average latency of under ten ms, while we see a rapid increase when the throughput surpasses 800 requests/second. When the throughput surpasses 1200 requests/second, the average latency approaches a full second. This is likely a result of putting too much load on the Corums scheduler, as method calls are being scheduled at a higher pace than the scheduler is able to execute them at.

Batch Size: 1024

Figure 5.2 shows the results of the experiment with a batch size of 1024. In this experiment, we see an even lower average latency when using low to moderate throughput settings. The reason for this is that the requests that are simply added to a batch and

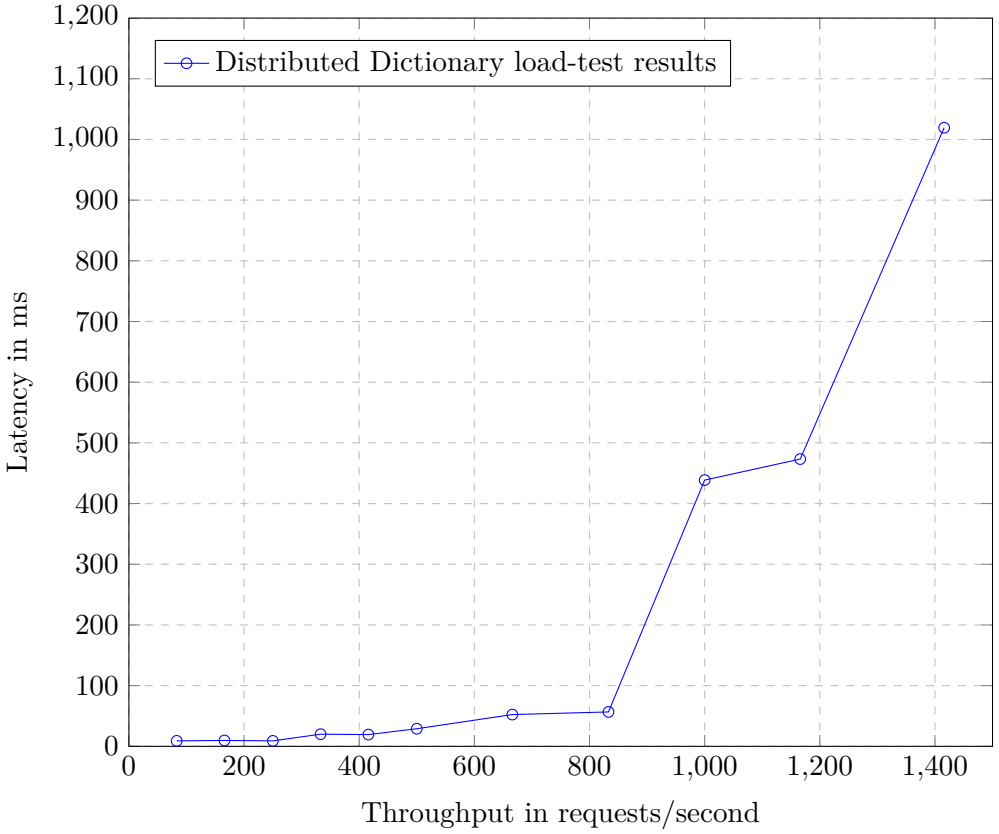

Figure 5.1: Latency vs. throughput with a batch size of 64. Each data point is the average of three experiments.

not pushed through a consensus round have *very* low latency, while the requests that result in a consensus round have higher latency. Thus, the larger the batch size, the lower the average latency, as a larger portion of the requests made will be very fast. In this experiment we see that the average latency is kept low until the throughput surpasses 10 000 requests per second. After that, it increases in a similar manner to the experiment with a batch size of 64, though it doesn't reach the same high values. The reason the average latency doesn't reach the same extremes in this experiment is that such a large portion of the requests are being batched, so even when the Corums scheduler is overloaded, the slow requests that result in a consensus round don't affect the average latency number as much.

In 2017, a Master's thesis [38] was done to investigate the use of Gorums [15] for implementing a fault-tolerant replicated service. This project is similar to ours in many ways, as they also implemented a consensus algorithm as the backbone of a replicated key-value store. Their solution used a Gorums-based implementation of the Raft consensus algorithm [16] to provide state replication and was implemented in the Go programming language. Comparing our performance efficiency results with the results they got shows that the latency at lower throughput settings are similar, but the latency in our Corums

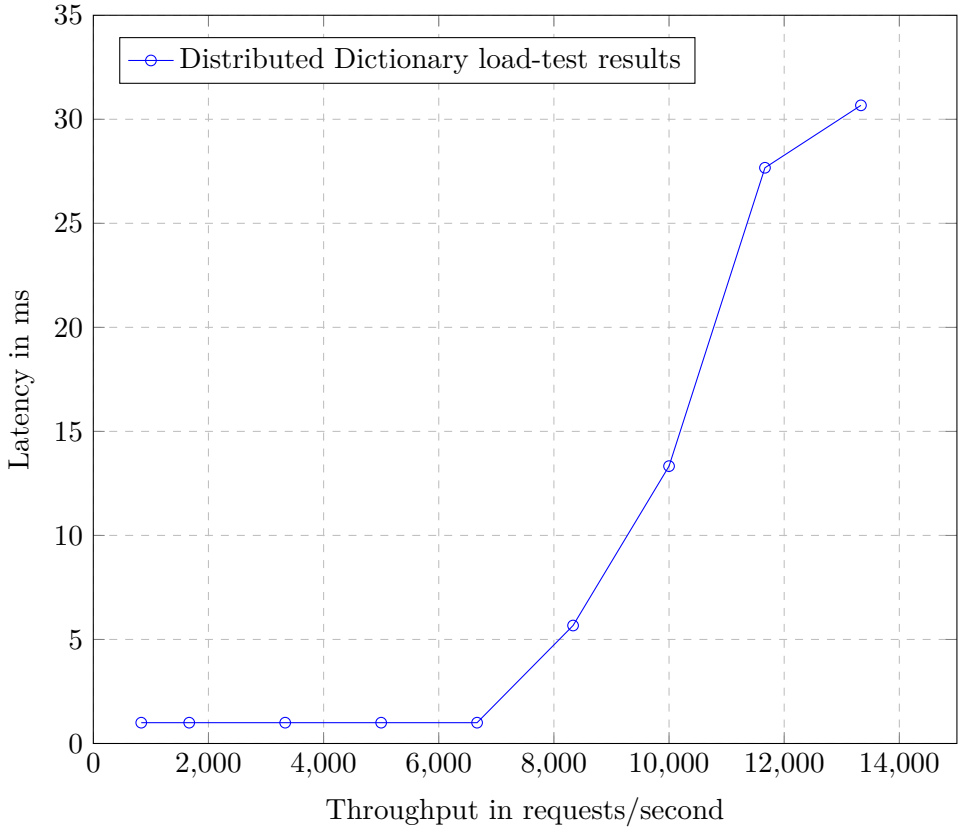

Figure 5.2: Latency vs. throughput with a batch size of 1024. Each data point is the average of three experiments.

Paxos implementation starts increasing at a substantially lower throughput setting than their Gorums Raft implementation does. Especially with the smaller batch size of 64, we see that their results show a stable low latency until the throughput reaches around 14 000 requests per second, at which point it starts increasing, though at a slower rate than in our measurements. There could be multiple reasons for these dramatic differences in performance, such as:

- Running on different hardware. The results from the Gorums Raft service was achieved running on machines in Amazon's Elastic Compute Cloud [39] with 4 virtual CPUs and 32.75 GB of RAM.
- Different software environment. Their system runs on the Amazon Linux AMI operating system and runs in the Go runtime.
- Different algorithm. Their system uses Raft, while our system uses Paxos. Though these algorithms are similar, they are not the same, and minor differences are amplified when running under high-load.
- Differences in application logic and the quality of our algorithm implementations.

- Differences in speed of the Gorums framework vs. the Corums framework. As the Gorums framework is built using multi-threading for concurrency, it will naturally handle high numbers of concurrent requests better than Corums.

Most likely, a combination of multiple or all of the above points contribute to the differences. More investigation would have to be done to pinpoint where the major efficiency problems in our system lie.

Chapter 6

Lessons Learned, Conclusion and Future Directions

In this chapter we explain some of the lessons we have learned during our work with this thesis, as well as what we would like to see be the future work on Corums and Distributed Dictionary. Finally, we conclude on the work we have done.

6.1 Lessons Learned

6.1.1 Learning a Framework With No Community

When choosing a framework for a software development project, the popularity and community of frameworks are considered important [40, 41]. Choosing a framework with an active community will aid in learning a new framework because there will be a lot of resources such as books, forums, video tutorials, etc. that provide useful information and examples of usage.

As Corums is a brand new framework under development, there is no community yet. Thus, there are no resources to be found and no community to ask for help. Making a Google search for similar uses of Corums, or any use of Corums for that matter, will not yield any results. This made a world of difference when developing Distributed Dictionary, even though we did get good help from our supervisors. It required us to investigate the internal code of Corums in more detail than we would normally would have to with any other framework. This is the nature of participating in research projects, so we were somewhat prepared for this going into it, but we learned exactly *how important* a framework's community is once we ran into circumstances that Corums hadn't been used for before.

6.1.2 Lack of Documentation

When starting our work with Corums, we quickly discovered that it lacked code documentation. As a general rule, we would have liked to see documentation for all of the Corums source code, but we especially believe it's important for the user-facing parts. Though the abstractions are recognizable and well-named, we missed having the ability to utilize the quick documentation feature of our IDE to get a short explanation of what something is and how to use it. With the lack of documentation, we ended up having to ask our supervisors for explanations, and though the responses came quickly and were good, it's a not viable replacement for proper code documentation.

6.1.3 Contributing to Corums

By making the contributions to the networking implementation that we described in Section 4.2.4, we understood the workings of Corums in a better, more detailed manner. Though this isn't strictly necessary to use the framework, understanding its inner workings made us more equipped to develop our system, as well as reasoning about what separates Corums from other frameworks.

6.1.4 Learning C#

Going into this project, we had little experience using the C# programming language. As programmers with Java as our "mother tongue", learning C# has been a good experience. The documentation that Microsoft produces is clear, the community is active and helpful and the development tools are good. In addition, the syntax of C# is recognizable and clear, leading to less time struggling with language barrier and more time to think about the actual logic.

6.2 Future Work

For technical optimizations to Corums, we would encourage focus on optimizing the scheduler to achieve higher performance efficiency. In our load-tests with smaller batch sizes, we had problems with high latency numbers at relatively low throughput settings. We believe these problems were caused by the Corums framework internally and would strongly encourage uncovering where possible bottlenecks and making the necessary optimizations. Though the nature of Corums being single-thread would lead us to expect lower performance than multi-threaded counterparts, we don't believe Corums

has achieved its full potential in terms of speed. We would also advocate for more testing done on the TCP communication of Corums, as it is a very crucial part of the framework which we believe has not gotten the focus it deserves.

To measure usability on a more accurate level, a large group of users should be given access to the framework, ideally observed when learning and then answer a questionnaire about their experience afterwards. The larger and more diverse the group of testers, the more accurate measurements we get. Try to get as detailed feedback as possible and take these into consideration for the further development of the framework.

If Corums aims to achieve industry adoption, there should be work done to encourage community growth by arranging workshops, starting forums, creating video tutorials, etc. Because a community is so important to developers when choosing a framework, new users of Corums should be encouraged to share knowledge and participate actively. We would also encourage publishing the source code in a public repository, allowing Corums' users to investigate the inner workings themselves and participate in deciding the future direction of the framework.

Finally, we would strongly encourage a greater focus on code documentation in the Corums source code going forward. Start by documenting all new code that is added and writing documentation for existing components when they are modified. Eventually, there should be made an effort by systematically going through the code base and verifying that all important components are well-documented.

There should be done more testing on Distributed Dictionary to investigate the correctness of our Paxos implementation. We would encourage implementing a specialized Distributed Dictionary client which could be used to insert randomized data at a high rate, then check the correctness of the dictionary afterwards. The client could also be utilized for testing the system in the event of the leader crashing. It should have a mechanism to detect that the leader it has communicated with has crashed, and then it should attempt to connect to a new server. There could be made changes to the server replicas to optimize the process of the client finding the new leader, where if the client tries to connect to a server who is not the leader, that server can tell the client who is.

Additional testing on the Distributed Dictionary Paxos implementation with focus on fault-toleration should be done. We recommend performing tests where participating servers are intentionally crashed or taken out of the network for a while during execution, then started back up again, to assess how/if it's able to catch up again after missing a number of consensus rounds.

Another feature we would have like to seen included in the Distributed Dictionary Paxos implementation is a cluster reconfiguration policy. Such a policy should handle situations

where a minority of participating servers crash or are having issues. The leader should keep track of the participants' performance by utilizing the failure detector explained in Section 4.2.3, and when it detects a server is not responding properly, it could perform a reconfiguration, meaning that it replaces the faulty server with a new one.

6.3 Conclusion

Corums claims to make the work of developers implementing consensus algorithms easier. To verify if this statement holds, we implemented Multi-Paxos with it. We also built an application layer around our Multi-Paxos implementation to assess whether Corums is usable for practical applications. Our experience is that Corums *does* genuinely simplify the implementation of the Paxos protocol. We found the provided abstractions useful and clear. In addition, Corums manages other aspects of distributed programming, such as configuring network connections and handling thread interleaving, so that we are allowed to focus on the details of implementing the Paxos protocol instead.

With the contributions we have made in this thesis, we have uncovered some of the areas were Corums is lacking and thus, should get more focus. Our contributions also show that there is good use for Corums in the distributed systems field by explaining usability and compatibility of the framework. In addition, having a practical application, such as ours, implemented using Corums will make it easier to implement similar systems in the future. We predict that some of the difficulties we have had in implementing our system will become easier in the future, as more people use the framework so that more practical examples are produced and best practices takes form.

List of Figures

2.1 A perfect/ideal Paxos instance. 11

3.1 Corums' event loop architecture. 20

4.1 Distributed Dictionary system architecture. 24
4.2 Overview of our Multi-Paxos implementation. 29
4.3 Failure detector execution flow. 30
4.4 The structure of the Corums networking implementation we implemented. 32

5.1 Latency vs. throughput with a batch size of 64. Each data point is the average of three experiments. 40
5.2 Latency vs. throughput with a batch size of 1024. Each data point is the average of three experiments. 41

List of Tables

A.1 Load-test results with a batch size of 64. 50
A.2 Load-test results with a batch size of 1024. 50

List of Algorithms

3.1 Stream subscription example. 16
3.2 Stream operators example. 17
3.3 Corums' combination of reactive and object-oriented programming. 18
3.4 Corums network stream example. 19
3.5 Sending a message with Corums. 19
4.1 Defining an API endpoint in an ApiController class. 25
4.2 Calling a method through the CorumsFacade. 26
4.3 Setting up an event handler for receiving prepare messages on the acceptor. 27

Appendix A

Experimental Data

This appendix contains the measurements made for the performance testing of Distributed Dictionary. This data is used to create the plots in Section 5.3.2. All latency values in the data tables are averages calculated from running three tests with the same configured throughput.

Throughput (commits/second)	Latency (ms)
83	9.00
166	9.67
250	9.00
333	20.00
416	19.33
500	29.00
666	52.33
833	56.67
1000	438.67
1166	473.33
1416	1019.33

Table A.1: Load-test results with a batch size of 64.

Throughput (commits/second)	Latency (ms)
833	1.00
1666	1.00
3333	1.00
5000	1.00
6666	1.33
8333	5.67
10000	13.33
11666	27.67
13333	30.67

Table A.2: Load-test results with a batch size of 1024.

Appendix B

Distributed Dictionary Source Code

The source code for the Distributed Dictionary service can be found in this Github repository.

Bibliography

[1] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].*, May 1998. URL https://www.microsoft.com/en-us/research/publication/part-time-parliament/. ACM SIGOPS Hall of Fame Award in 2012.

[2] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001. URL https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[3] Hein Meling and Leander Jehl. Tutorial Summary: Paxos Explained from Scratch. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, editors, *Principles of Distributed Systems*, volume 8304, pages 1–10. Springer International Publishing, Cham, 2013. ISBN 978-3-319-03849-0 978-3-319-03850-6. doi: 10.1007/978-3-319-03850-6_1. URL http://link.springer.com/10.1007/978-3-319-03850-6_1. Series Title: Lecture Notes in Computer Science.

[4] Scott Addie and Tom Dykstra. Create web APIs with ASP.NET Core, February 2020. URL https://docs.microsoft.com/en-us/aspnet/core/web-api/. Library Catalog: docs.microsoft.com.

[5] IBM Global Services. Improving systems availability, August 1999. URL http://www.cs.cmu.edu/~priya/hawht.pdf.

[6] Per Christensson. Scalable definition, Januaury 2011. URL https://techterms.com/definition/scalable.

[7] Coinbundle Team. Consensus algorithms, September 2018. URL https://medium.com/coinbundle/consensus-algorithms-dfa4f355259d.

[8] Mark Brooker. Leader election in distributed systems, 2019. URL https://aws.amazon.com/builders-library/leader-election-in-distributed-systems/. Library Catalog: aws.amazon.com.

[9] Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. The ph accrual failure detector. *Unknown*, 01 2004. doi: 10.1109/RELDIS.2004.1353004.

[10] Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43, 07 1999. doi: 10.1145/226643.226647.

[11] Nuno Santos and André Schiper. Optimizing paxos with batching and pipelining. *Theoretical Computer Science*, 496:170–183, 07 2013. doi: 10.1016/j.tcs.2012.10.002.

[12] Thomas Stidsborg Sylvest, Hein Meling, Leander Jehl, and Veronica Estrada-Galiñanes. Corums - simplifying implementation of consensus algorithms. *Unpublished*, 12 2018.

[13] Yanhong Liu, Scott Stoller, and Bo Lin. High-level executable specifications of distributed algorithms. In *Unknown*, volume 7596, pages 95–110, 10 2012. doi: 10.1007/978-3-642-33536-5_11.

[14] What is .NET? An open-source developer platform., 2020. URL https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet. Library Catalog: dotnet.microsoft.com.

[15] 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)*, pages 2380–2385, 2017.

[16] D. Ongaro, J.K. Ousterhout, D.F. Mazières, M. Rosenblum, and Stanford University. Computer Science Department. *Consensus: Bridging Theory and Practice*. Stanford University, 2014. URL https://books.google.no/books?id=xfLynQAACAAJ.

[17] Sebastian Pedersen, Hein Meling, and Leander Jehl. An analysis of quorum-based abstractions: A case study using gorums to implement raft. In *Unknown*, pages 29–35, 07 2018. doi: 10.1145/3231104.3231957.

[18] Clement Escoffier. 5 Things to Know About Reactive Programming, June 2017. URL https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming/.

[19] dotnet bot. Task class (system.threading.tasks), 2020. URL https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task. Library Catalog: docs.microsoft.com.

[20] Bill Wagner. lock statement - C# reference, February 2020. URL https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/lock-statement. Library Catalog: docs.microsoft.com.

[21] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction To Reliable And Secure Distributed Programming*. Springer-Verlag Berlin Heidelberg, 2011. URL https://www.springer.com/gp/book/9783642152597.

[22] Srikanth Sastry and Scott Pike. Eventually perfect failure detectors using add channels. In *Unknown*, pages 483–496, 08 2007. doi: 10.1007/978-3-540-74742-0_44.

[23] Why Docker? | Docker, 2020. URL https://www.docker.com/why-docker. Library Catalog: www.docker.com.

[24] .NET Core SDK - Docker Hub, 2020. URL https://hub.docker.com/_/microsoft-dotnet-core-sdk/.

[25] ASP.NET Core 2.1/3.1 Runtime - Docker Hub, 2020. URL https://hub.docker.com/_/microsoft-dotnet-core-aspnet/.

[26] Overview of Docker Compose, June 2020. URL https://docs.docker.com/compose/. Library Catalog: docs.docker.com.

[27] ISO/IEC 25010. ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. Technical report, International Organization for Standardization, 2011. URL https://www.iso.org/standard/35733.html.

[28] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering*, 43(12):1125–1143, 2017.

[29] Jan Kończak, Nuno Filipe de Sousa Santos, Tomasz Żurkowski, Paweł T. Wojciechowski, and André Schiper. Jpaxos: State machine replication based on the paxos protocol. *Unknown*, page 38, 2011. URL http://infoscience.epfl.ch/record/167765.

[30] Nuno Santos, Jan Kończak, and Tomasz Zurkowski. JPaxos/JPaxos, April 2020. URL https://github.com/JPaxos/JPaxos. original-date: 2011-05-16T07:00:54Z.

[31] Vaida. C# vs Java Tutorial: Find Out Difference Between C# and Java, May 2019. URL https://www.bitdegree.org/tutorials/c-sharp-vs-java/. Library Catalog: www.bitdegree.org Section: Comparisons.

[32] C# vs Java: Differences you should Know, June 2020. URL https://hackr.io/blog/c-sharp-vs-java. Library Catalog: hackr.io.

[33] Google. The Go Programming Language, 2020. URL https://golang.org/.

[34] Naveen Ramanathan. Goroutines - Concurrency in Golang, July 2017. URL https://golangbot.com/goroutines/. Library Catalog: golangbot.com.

[35] Microsoft. What is NuGet and what does it do?, 2020. URL https://docs.microsoft.com/en-us/nuget/what-is-nuget. Library Catalog: docs.microsoft.com.

[36] Stack Overflow. Stack Overflow Developer Survey 2020, 2020. URL https://insights.stackoverflow.com/survey/2020/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2020. Library Catalog: insights.stackoverflow.com.

[37] Apache JMeter - Apache JMeter™, 2020. URL https://jmeter.apache.org/.

[38] Sebastian Mæland Pedersen. A practical analysis of the gorums framework: A case study on replicated services with raft. Master's thesis, University of Stavanger, 2017. URL http://hdl.handle.net/11250/2455424.

[39] Amazon. Amazon EC2, 2020. URL https://aws.amazon.com/ec2/. Library Catalog: aws.amazon.com.

[40] Symfony. Ten criteria for choosing the correct framework, 2020. URL https://symfony.com/ten-criteria. Library Catalog: symfony.com.

[41] Christian Varisco. How to choose a framework | Hacker Noon, October 2016. URL https://hackernoon.com/how-to-choose-a-framework-ea8b5b1e1f44. Library Catalog: hackernoon.com.