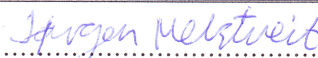




University of
Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/Specialization: Computer Science	Spring semester, 2021 Open
Writer: Jørgen Melstveit	 (Writer's signature)
Faculty supervisor: Leander Nikolaus Jehl External supervisor(s): Thomas Stidsborg Sylvest	
Thesis title: Implementing PBFT using Reactive programming and asynchronous workflows	
Credits (ECTS): 30	
Key words: Practical Byzantine Fault Tolerance, Cleipnir, Consensus Algorithms, Distributed Systems, Asynchronous Programming, Reactive Programming, Protocol Descriptions	Pages: 91 + enclosure: Source code on Github Stavanger, July 11, 2021 Date/year

Implementing PBFT using Reactive programming and asynchronous workflows

Jørgen Melstveit

June 2021

Contents

1	Introduction	2
1.1	Contributions	3
1.2	Outline	3
2	Programming Models	5
2.1	Asynchronous Programming	5
2.1.1	Async/Await	6
2.2	Reactive Programming	7
2.2.1	Reactive X	8
3	Cleipnir	10
3.1	Cleipnir Overview	10
3.2	Cleipnir Reactive Programming	13
3.3	Cleipnir Persistent Programming	14
4	Practical Byzantine Fault Tolerance	18
4.1	Introducing Practical Byzantine Fault Tolerance	18
4.2	System Model	19
4.3	Detailed Protocol Operations	20
4.4	Checkpointing	22

CONTENTS

4.5	View-change	23
5	Related Work	25
5.1	Cleipnir - Framework Support for Fault-tolerant Distributed Systems . . .	25
5.2	Implementing a Distributed Key-Value Store Using Corums	26
6	Design	27
6.1	Network Architecture	27
6.2	Overview of Workflow	29
6.3	Code structure	30
6.3.1	Protocol Objects	30
6.3.2	Other functionalities	30
6.3.3	JSON Serialization Problem	31
6.3.4	Notable Files	32
6.4	Persistent vs Ephemeral	34
7	Implementation	37
7.1	Design Choices	37
7.2	Workflow Details	39
7.2.1	Protocol Workflow Implementation	39
7.2.1.1	Starting protocol instance	39
7.2.1.2	Pre-Prepare phase	40
7.2.1.3	Prepare phase	43
7.2.1.4	Commit Phase	44
7.2.1.5	Protocol Workflow Evaluation	45
7.2.2	Checkpoint Implementation	49

CONTENTS

7.2.2.1	Initialize Checkpoint Certificate	49
7.2.2.2	Checkpoint Listener Workflow	50
7.2.2.3	Initiate Garbage Collection	51
7.2.2.4	Checkpoint Workflow Evaluation	52
7.2.3	View-change Implementation	53
7.2.3.1	Starting a View-Change	54
7.2.3.2	View-Change functionality	56
7.2.3.2.1	View-Change Listener Workflow	59
7.2.3.2.2	New-View Workflow	60
7.2.3.3	View-Change Evaluation	64
7.3	Client	65
8	Discussion	67
8.1	Protocol Abstraction	67
8.2	Asynchronous workflow	68
8.3	Usage of Cleipnir	69
8.3.1	Reactive Operators	69
8.3.2	Persistency	70
9	Conclusion	72
9.1	Lessons Learned	72
9.1.1	Consensus algorithms	72
9.1.2	Asynchronous Programming	72
9.1.3	Reactive Programming	73
9.1.4	Cleipnir	73
9.2	Future Work	74

CONTENTS

9.3 Conclusion	76
Appendix A Practical Byzantine Fault Tolerance (PBFT) Implementation Source Code	79
Bibliography	80

Abstract

Consensus algorithms are notorious for being both difficult to understand and even harder to implement. Several frameworks and programming paradigms have been introduced to help make consensus algorithms easier to design and implement. One of these frameworks is the .NET Cleipnir framework which primarily focuses on making it simpler to develop a persistent consensus algorithm. In addition, Cleipnir supports functionality that makes both asynchronous and reactive programming paradigms easier for a developer to utilize in their implementation. We want to determine if the Cleipnir framework and the related programming paradigms can help design a simple and understandable consensus algorithm. To accomplish this task, we create a Practical Byzantine Fault Tolerance implementation that has its protocol workflow run as orderly and synchronous as possible using the Cleipnir framework and the aforementioned protocol paradigms. Furthermore, we evaluate each of the previously mentioned tools to ascertain how they benefit and hinder our implementation. We discover that the benefits heavily outrank the disadvantages for both programming paradigms and works well together. We conclude that the Cleipnir framework does provide helpful tools for the implementation of consensus algorithms. We further learn that the algorithm's complexity can heavily affect the level of simplicity that can be provided to the algorithm workflow without the loss of functionality.

Acknowledgement

I want to thank my supervisor Professor Leander Nikolaus Jehl, for providing consistent feedback and guidance throughout our thesis. I would also like to express my gratitude forwards Thomas Stidsborg Sylvest, who helped us learn the basics of the Cleipnir framework by sharing his expertise and answering any additional questions we had during our thesis.

Acronyms

API Application Programming Interface

APM Asynchronous Programming Model

EAP Event-based Asynchronous Pattern

FCFS First Come First Serve

IP Internet Protocol

JSON JavaScript Object Notation

LINQ Language Integrated Query

MAC Message Authentication Code

PBFT Practical Byzantine Fault Tolerance

SQL Structured Query Language

TAP Task-based Asynchronous Pattern

TCP Transmission Control Protocol

Chapter 1

Introduction

Systems today are required to be both efficient, secure, and reliable. Due to these factors, most firmware and software today are organized over multiple systems in what we call a *distributed system* [1], [2, p. 16]. In distributed systems, network nodes are required to share and collaborate so that the systems can agree on an overall state of the system. This state must remain consistent for the systems even in the event of failure, or in some cases, malicious intent. A distributed system must be able to act as if it is a single system, even when in reality it is composed of multiple systems [2, p. 18]. Advanced and technical consensus algorithms are currently being used to handle this functionality. However, most consensus algorithms are known for being difficult to comprehend and can be even more demanding to implement due to the unreliable nature of distributed networks [2, p. 459], [3, p. 13]. Because of this, alternative ways to describe and implement existing consensus algorithms are being discussed.

The University of Stavanger has previously published work that implements popular consensus algorithms, such as Paxos and Raft [4], in a simplified manner using frameworks that support reactive programming. In particular, "Cleipnir - Framework Support for Fault-tolerant Distributed Systems" [5] and "Implementing a Distributed Key-Value Store Using Corums" [3] uses the .NET framework now known as *Cleipnir* [6]. Cleipnir is a .NET framework designed to help make implementations for consensus algorithms simpler for the developer. These two previously mentioned works are predecessors for this thesis which intends to use the Cleipnir framework to implement another popular consensus algorithm to analyze Cleipnir ability further to simplify the implementation of consensus algorithms.

The goal for this thesis is to use the Cleipnir framework to implement the Practical Byzantine Fault Tolerance consensus algorithm using functionality from both asynchronous programming and reactive programming [7]. The desired PBFT implementation should be devised using `async/await` functionality existing in the .NET framework [8] and reactive event handling, both which Cleipnir supports. The goal for the PBFT implementation is to use these tools to create a workflow that is simple so that the source code can both be easily read but also easily recreated. To achieve these goals, we are looking into Cleipnir current support for reactive programming. We also look at the current workflow

1.1 Contributions

of modern asynchronous programming for .NET. We also examine the PBFT algorithm and present a detailed summary of all of its processes. Additionally, Cleipnir hybrid persistency functionality is reviewed. In the end, the question is whether Cleipnir and these programming paradigms have the sufficient support required to accomplish these goals. We will also see how advantageous reactive programming paradigm and asynchronous programming are when designing a consensus algorithm.

1.1 Contributions

To tackle the problem, we implemented a simple PBFT implementation that primarily uses `async/await` asynchronous programming and reactive event handlers provided by the Cleipnir framework to design the normal workflow of PBFT inside a single function. The source for our PBFT implementation can be found at [7]. The normal workflow function is designed to follow the protocol description as closely as possible. To accomplish this goal, the PBFT consensus algorithm was studied in great detail. In addition, we had to implement the network layer for the PBFT implementation using .NET asynchronous socket programming [9], [10]. In this thesis, we have also looked at how both asynchronous programming and reactive programming have affected the simplicity of the protocol code and can conclude both positive and negative aspects for the programming models. To the best of our ability, we designed the PBFT application to utilize Cleipnir's tools as much as possible. Although our current implementation does not fully support persistency, we have taken steps to ensure at least that the protocol objects and protocol-related functionality are designed with persistency in mind. We believe that the thesis does present some helpful feedback for future development of the Cleipnir framework.

1.2 Outline

- In **Chapter 2** we briefly describe the background information in regards to this thesis. This includes information in regards to asynchronous programming and reactive programming
- In **Chapter 3** we make an introduction to the Cleipnir framework. This includes describing the intended use-case for Cleipnir and summarize its core functionalities that are potentially helpful for implementing consensus algorithms.
- In **Chapter 4** we describe the PBFT algorithm. This includes introducing the main goals and processes of the consensus algorithm. We also briefly describe concepts used by or related to the algorithm. Finally, a detailed summary of all the operations taking place in the algorithm is presented.
- In **Chapter 5** we introduce previous work in regards to the Cleipnir framework and other related work that are similar to this project.
- **Chapter 6** introduces an overview of our application. We first give a short summary of how the network is set up for the PBFT implementation. Then we go more

in-depth about how we've structured the code for the implementation. Finally, we describe how the application is divided into separate segments based on whether or not the segment uses Cleipnir to persist its data.

- **Chapter 7** gives a detailed explanation of our PBFT implementation. We start by first presenting our choices in design to accomplish our main objectives. Then the normal workflow implementation is described in detail. In addition, we discuss how the implementation handles view-changes and checkpoints. We describe for each workflow how asynchronous programming and Cleipnir reactive programming have helped or hindered simplifying the code for our implementation. Finally, we discuss some drawbacks to our design.
- **Chapter 8** gives a summary of all of the benefits and disadvantages we encountered for each of the tools and designs we used in our PBFT implementation.
- **Chapter 9** is the last chapter and it contains a conclusion for the given PBFT implementation based on the initial goals. Furthermore, we also summarize our results, discuss the knowledge we accumulated during the thesis, and suggest future work.

Chapter 2

Programming Models

Information about the asynchronous programming and reactive programming models are introduced in this chapter. This includes their intended use cases and general workflow. The asynchronous programming section mentions several design patterns used for asynchronous operations. We will mainly concentrate on the `async/await` model [8]. The reactive programming section covers information about ReactiveX [11] which is the cornerstone for all Rx-driven implementations.

2.1 Asynchronous Programming

Asynchronous programming is a programming technique designed to handle a common problem that sometimes occurs in synchronous programming. Synchronous programming always blocks the execution until the previous line of code is handled. A synchronous program forces the program to finish a single operation in the program before moving on to the next operation. However, blocking the execution thread usually leads to scalability issues, latency issues and generally results in an awful user experience. Meaning synchronous programming is not optimal for operations that require a long execution time. Especially if the operation itself spends most of its time waiting for a result, examples of such actions would be database requests or I/O bound operations [12], [13]. Keep in mind that asynchronous programming for different programming languages usually has similar workflows. However, the naming conventions for identical operations may differ. In this thesis, the terminology used for asynchronous programming follows the ones used in the .NET framework.

Asynchronous programming, as the name implies, is designed to run operations asynchronously. In the asynchronous programming model, operations are divided into a set of tasks. These tasks perform the assigned operations whenever the scheduler has resources it can delegate to them. However, the task created does not block the main thread, instead, the main thread continues with the next operations [8], [12], [13]. The task has a reference to an awaiter that has information on the current state of the task. Eventually, the asynchronous operation finishes, and the result is available in the awaiter for the main

2.1 Asynchronous Programming

thread to collect. Not all tasks need to return a result necessarily. It is possible to run non-returning asynchronous operations in tasks as well. Nevertheless, a task must always return an awaiter so that the main thread has reference to all relevant information for the asynchronous task [13].

Normally, the main thread needs to receive the result of the asynchronous operation before reaching specific parts of the program that requires the result to function correctly. Asynchronous programming supports this functionality by allowing the designer to specify to the awaiter that the program is to wait at this point until the asynchronous operation is finished. This still does not block the main thread, meaning other tasks can be performed in the background, unlike synchronous programming. Additionally, asynchronous programming has the benefit that the operation can be initialized earlier and be worked on by the main thread while going through the main thread operations to the point where the result is expected. This means asynchronous programming could avoid bottlenecks that occur in synchronous programming, thereby making asynchronous programming more responsive of the two programming models [13], [14]. For this reason, asynchronous programming has become the preferred programming model for designing user interfaces since it is crucial to avoid potentially blocking user input when at the same time, other primary tasks are performed [12], [15, p. 214]. Server design is another example where asynchronous design is preferred as it handles many requests easier than a server with synchronous design [8], [12].

Asynchronous programming usually follows one or more of these three design patterns:

- Asynchronous Programming Model (APM)
- Event-based Asynchronous Pattern (EAP)
- Task-based Asynchronous Pattern (TAP)

TAP is the most used design pattern and is the model used by the `async/await` workflow [8], [13].

Asynchronous programming should not be confused with parallel programming, as asynchronous methods do not create new threads. It instead runs on the current thread whenever the scheduler has resources ready, and the operation itself is ready to progress. Therefore, the work required to create new threads and a lot of the work to keep the threads consistent can be omitted [14].

2.1.1 Async/Await

.NET has long had support for asynchronous programming [16]. However, before the `async/await` workflow became normalized, programming asynchronously was quite difficult and even worse for others to read [14], [16]. The old workflow consisted of a lot of nested callback functions, which is a struggle to manage properly. Today managing this kind of structure is referred to as *callback hell* [17, p. 1-2], [5, p .2].

2.2 Reactive Programming

As previously mentioned, the `async/await` workflow follows the TAP abstraction [14]. The `async/await` workflow, therefore, consists of creating a task that performs the asynchronous operation. Then the original process that created the asynchronous task marks where the result of the task needs to be returned in the workflow. If the task is not finished when it reaches the marked area in the workflow, the process waits at this point until the result is ready. The `async/await` workflow consists of three steps for the programmer. The first step is to assign the `async` modifier to a function to mark it as an asynchronous function. This allows asynchronous calls to be made inside the chosen function. The second step is to make an asynchronous call. Lastly, specify the `await` operator for the awaiter for the asynchronous task to determine where in the workflow the result is obtained [8], [12], [13]. It is important to remember that the `await` operator can only be used in a function marked with the `async` modifier. The traditional asynchronous operators have to be used instead of the `async/await` workflow when making asynchronous calls inside synchronous functions [8], [14].

In Listing 2.1 we can see a practical example of the `async/await` workflow. The code in Listing 2.1 is the asynchronous process that is responsible for having a chosen `Socket` object connect to a designated Internet Protocol (IP) address. The `IPEndPoint` object being the reference to the chosen IP address. In order for the `Connect` function to be marked as an asynchronous function it has a `async` modifier. `Connect` returns a `.NET Task` object of type `boolean`, meaning the function returns a reference to the active `Connect Task` which returns a `boolean` value once the `Task` is completed. In this case the `Connect` function returns `true` if the socket succeeds in connecting to the IP address, otherwise it returns `false`. The asynchronous operation performed inside the `Connect` function is the `ConnectAsync` function which is called by the socket object. As we want to avoid the function returning the result before the asynchronous operation is finished, the `await` operator is used to have the `Task` wait for the `ConnectAsync` asynchronous operation to finish.

```
1 public static async Task<bool> Connect(Socket sock, IPEndPoint endpoint)
2 {
3     try
4     {
5         await sock.ConnectAsync(endpoint);
6         return true;
7     }
8     catch (Exception e)
9     {
10        Console.WriteLine("Failed to connect to endpoint: " + endpoint.Address);
11        Console.WriteLine(e);
12        return false;
13    }
14 }
```

Listing 2.1: Example of `async/await` workflow

2.2 Reactive Programming

Reactive programming is a programming paradigm whose main focus is to change the state of the program in response to some outward changes [6], [18]. Reactive programming follows an event-driven workflow. An event can be triggered from one part of the

2.2 Reactive Programming

system, and when the other part of the system receives this event, it alters the state of the system in response. Reactive programming works hand in hand with asynchronous event-based programming, which was previously mentioned briefly in Section 2.1 [19, p. 2-3]. Reactive programming is commonly used to handle a continuous stream of asynchronous data [20]. Currently, there exists a lot of support for Reactive programming. Specifically, the library Reactive X [11] has presented a general Application Programming Interface (API) [21] for implementing the core concepts of reactive programming. As a result, today, there exist a lot of reactive extensions for multiple programming languages. Rx.NET [22] is the official .NET reactive extension. Cleipnirs has implemented its own reactive extension that closely resembles Rx.NET. The main difference between the two is that Cleipnirs reactive layer supports persistency but lacks reactive operators that Rx.NET does support [6]. Although Cleipnir and Rx.NET vary somewhat from the general API, the general workflow remains the same. Therefore we will introduce the main concepts of Reactive X in this section. Details specific for Cleipnir are instead presented in the upcoming Chapter 3.

2.2.1 Reactive X

ReactiveXs workflow can be easily summarized with the following tasks [23]

1. Start an asynchronous operation that will perform some work and eventually return it
2. Transform the asynchronous operation as an Observable object
3. Use reactive operators to transform/filter the resulting data.
4. Observers subscribe to the Observable and waits for the Observable to return the data

An observable object follows a similar structure to an enumerable object, where the main difference between an enumerable and an observable object is their method of accessibility. An enumerable object will give the next object in storage whenever asked for it. In other words, the program will dictate when the next entry is collected. In an Observable object, the next result is instead only pushed to its subscriber whenever the result is ready. The program has no control over when the next entry will be ready as it is waiting for an asynchronous operation to complete [20], [23], [24], [19, p. 15]. Observables, like enumerable, support the use of Language Integrated Query (LINQ) queries on its resulting data. LINQ add additional operators for filtering and transforming the resulting data into new enumerables [20], [19, p. 3-4], [15, p. 208].

Traditionally, the implementation is expected to incorporate the following functions for its observer object.

- OnNext

2.2 Reactive Programming

- `OnError`
- `OnCompleted`

`OnNext` is the function that handles each new incoming event emitted by the `Observable`. `OnError` is the function that is called if an error occurs within handling one of the emitted events. `OnCompleted` is the function that is called when the observable is finished and will no longer emit any new events [23].

In some implementations, the `Observable` and observer functionality are merged together into an object referred to as a *subject*. A subject object acts as a bridge of sorts between the observer and the observable, where its primary usage is to simplify the workflow for reactive programming. A subject has the ability to subscribe to an observable, just like an observer. However, unlike an observer, a subject can also re-emit events already processed in the observable, and be used for emitting new events to the observable. Eventually, all the items emitted by the subject is handled by the subject, making the programming workflow a lot simpler compared to its traditional style [25]. Cleipnir supports subject object in its implementation, however, the objects are not called `subject` in Cleipnir's implementation but are instead called `Source` objects.

Chapter 3

Cleipnir

Cleipnir is a .NET framework primarily designed for aiding in implementing consensus algorithms. Specifically, the framework's main contribution is assisting developers with creating persistent distributed systems. Prior to this thesis, Cleipnir and its predecessor Corums, have been used to implement two consensus algorithms, namely Paxos [3, p. 32-38] and Raft [5, p. 13-15]. Cleipnir is designed to support and work with the three following programming paradigms [5, p. 5]:

- Reactive Programming
- The Async/Await Model
- Persistent Programming

Two of these programming paradigms were already presented in Chapter 2. Therefore, only the Persistent Programming paradigm is introduced in this chapter. The `async/await` model used in Cleipnir is the official implementation from the .NET framework [8]. As mentioned in Section 2.2, Cleipnir uses a custom-built reactive framework, and this framework is discussed in detail in this chapter.

The information presented in this chapter is based on the Cleipnir paper [5], its current documentation [6] and from informative conversations with the creator of Cleipnir, Thomas Stidsborg Sylvest.

3.1 Cleipnir Overview

There are three main tools that Cleipnir provides developers to help design their application. These three tools are:

- Persistent Synchronous Scheduler

3.1 Cleipnir Overview

- Storage Engine
- Object Store
- Reactive Programming Layer

Cleipnir uses an inbuilt event-driven **scheduler** that follows a single-threaded structure similar to the JavaScript scheduler [26], [5, p. 7]. The scheduler schedules incoming tasks in a queue structure, meaning the ordering follows a First Come First Serve (FCFS) [27] approach. Each task in the queue is executed sequentially using only a single thread, which in theory allows the program to avoid common threading issues [5, p. 7].

The **storage engine** is responsible for the actual storage procedure. It is responsible for performing both the serialization and the deserialization process to each state object that is to be persisted. The details regarding setting up object information for the serialization and deserialization for a state object is presented in Section 3.3. Cleipnir uses different storage engines correlated to the kind of storage that is used to store the data. Cleipnir currently supports these three storage engines:

- Memory Storage
- Simple File Storage
- Relational Database Storage

[5, p. 10, 12]

The memory storage stores the persisted data directly into memory. The simple file storage stores the persisted data in a single text file. The relation database storage stores the persisted data into a Microsoft SQL Server [28], [5, p 10-12].

The Cleipnir serialization process follows a graph-like structure. The original object graph that is to be serialized is called a **Roots** object. In order to accomplish this, the **Roots** graph object to the persisted object is connected to the graph object through pathways leading to the references that are also to be persisted [5, p. 10].

The **object store** is responsible for accessing the storage engine. The object store uses the storage engine whenever the application needs to restore some previously persisted data, using the storage engine to persist a new object or updating existing object records in the persisted memory. The object store is also responsible for detecting changes done to any state variables that are registered to be persisted. The object store uses a statemap to keep track of records for each of the state variables that are to be persisted and stored by the storage engine [5, p. 11].

Listing 3.1 shows a short example of how to use the object store to cache an object into the storage engine and then restore that object after the data is lost in the application. First, both the storage engine and the object store are initialized, where the type of storage engine used for this example is the simple file storage. Then the object store uses

3.1 Cleipnir Overview

the `Attach` function to register the request object to the object store. The object store now has a `Roots` entry for the request object. In this example, the `Persist` function is used to serialize and store the objects currently registered in the object store. Object store only persists objects that have either not been cached before or if any changes have affected the object when `Persist` is called. In this example, only the request object with its system references are registered by the object store. Therefore only the request object is persisted. In the example, the object store is intentionally reset by assigning it the value `null` to test that the request object is properly persisted. The object store is then reloaded by attaching it with the previous storage engine that we know has the request object in its statemap. Therefore, using the `Resolve` function, we restore the request object and assign it to a new variable. Since the original request object was never tampered with, we know that the initial request and the newly resolved request should be equal. Most of the functionality shown in Listing 3.1 is performed behind the scenes, and a developer rarely has to attach an object to the object store directly.

```
1  _storage = new SimpleFileStorageEngine("PersistentStorage.txt");
2  _objectStore = ObjectStore.New(_storage);
3  (_pri, _pub) = Crypto.InitializeKeyPairs();
4  var currentTime = DateTime.Now.ToString();
5  Request req1 = new Request(1, "Hello World!", currentTime);
6  req.SignMessage(_pri);
7
8  _objectStore.Attach(req1);
9  _objectStore.Persist();
10 _objectStore = null;
11 _objectStore = ObjectStore.Load(_storage, false);
12 Request req2 = _objectStore.Resolve<Request>();
```

Listing 3.1: Object Store example

The scheduler and the object store operate independently from each other. In order for an application to take advantage of both of these tools, Cleipnir has an execution engine that utilizes both tools to the best of their abilities. The Cleipnir execution engine's overall architecture is constructed so that the scheduler and the object store can be used together and collaborates within a single mechanism. Using the execution engine, the developer can specify the tasks that are to be executed by the scheduler and use the object store to persist the state of the application during certain parts of the execution. The execution engine uses what is known as `Sync` points to determine when to call the `Persist` function in the object store. The developer has to manually add these points in areas where the state can become corrupt if not crucial information is persisted when a system crash occurs. This is important for persisted consensus algorithms as it needs a stable state to reboot to even if an active process was cut mid-execution. This would otherwise guarantee significant consequences for the overall state of the distributed system. By default, if the scheduler does not have any tasks in its queue and is not working on any existing tasks, then it should also call the `Persist` function so that it can save changes in the state during a silent period. Listing 3.2 shows an example of how to initialize the execution engine and how to schedule an operation [5, p. 11].

3.2 Cleipnir Reactive Programming

```
1 var storageEngine = new SimpleFileStorageEngine(".PBFTStorage"+paramid+".txt", false);
2 scheduler = ExecutionEngineFactory.StartNew(storageEngine);
3 scheduler.Schedule(() =>
4 {
5 ...
6 });
```

Listing 3.2: Execution engine example

3.2 Cleipnir Reactive Programming

The Cleipnir framework has a custom-made reactive layer that follows most of the functionality provided by the Reactive X API. However, the basic functionality introduced in Section 2.2 is mostly hidden, and the overall workflow is simplified to make it easier for developers to use the reactive framework. This implementation uses a **Stream** object to replicate the data returned by the respective observers and operators. The **Stream** is similar to an observable object. Cleipnir reactive layer supports less reactive operators compared to most other current reactive implementations. However, the current operators that exist also support persistent programming, meaning the data stream and the scheduled operations are not lost if the system crashes during an operation. Traditional LINQ commands do not work on the **Stream** object. Instead inbuilt LINQ statements are available for the reactive **Stream** object to use. Cleipnir reactive operators can by design be chained together just like the majority of reactive operators in other frameworks. The main difference being that Cleipnir's reactive operators and LINQ operators result in a new **Stream** object instead of a new observable or a new enumerable. It is possible to create and handle a lot of the consensus algorithm workflow within a few lines of code by simply chaining reactive operators together. Listing 3.3 shows an example of chaining reactive operators using Cleipnir's reactive framework. The objective here is to get the first valid pre-prepare message emitted to the observable. For a pre-prepare message to be considered valid, it must pass all **Where** clauses. The **Next** operator at the end of the chain returns the resulting prepare message [5, p. 6, 8, 13], [29] The **Merge** operator is used to listen for incoming items from the **ShutdownBridgePhase** Source object. More information in regards to the **Merge** operator is discussed in Chapter 7

```
1 var preprepared = await MesBridge
2     .Where(pm => pm.PhaseType == PMessageType.PrePrepare)
3     .Where(pm => pm.Digest != null &&
4         pm.Digest.SequenceEqual(digest))
5     .Where(pm => pm.Validate(
6         Serv.ServPubKeyRegister[pm.ServID],
7         Serv.CurView,
8         Serv.CurSeqRange)
9     )
10    .Merge(ShutdownBridgePhase)
11    .Next();
```

Listing 3.3: Example of chaining Cleipnir reactive operators

Cleipnir supports reactive subject functionality. However, subject objects are instead referred to as **Source**. The user can emit items to the **Source** object, and any observer

3.3 Cleipnir Persistent Programming

linked to the `Source` object receives the response item. The `Source` objects are required to be used for the developer to access and interact with the reactive layer in Cleipnir. Listing 3.4 shows an example of how to initialize, emit and wait for incoming events in regards to the `Source` object. The `await reqbridge.Next()` makes sure that the resulting variable `req` receives the `Request` emitted to the `Source` object [5, p. 8].

```
1 Source<Request> reqbridge = new Source<Request>();
2 reqbridge.Emit(new Request(1, "Hello World!", DateTime.Now.ToString()));
3
4 Request req = await reqbridge.Next();
```

Listing 3.4: Source object example

3.3 Cleipnir Persistent Programming

A system that follows the persistent programming paradigm will regularly save the information for the program state while the program is running. Persistent programming makes it possible to design systems that can quickly restore their program state in the case of a system reboot [6], [5, p. 6]. Consensus algorithms can take great advantage of this programming paradigm as systems in the network are likely to crash eventually. With persistent programming, it is theoretically simple for a system to recover its data and rejoin the distributed network. Unfortunately, the state of the system is likely to still be somewhat behind the other systems when compared directly to the other working systems, even if all of the previous data is recovered.

Cleipnir supports easy to use hybrid persistent programming. Hybrid persistent programming allows the developer to freely choose which data is to be persistable. In this way, it is possible to avoid storing unnecessary information that would slow down the process immensely [5, p. 9-10]. Listing 3.5 and Listing 3.6 show an example of the workflow needed for an object to be serialized and deserialized to and from persistent memory. For an object to become visible to the storage engine, the object needs first to inherit either the `IPersistable` or the `IPropertyPersistable` interface. `IPersistable` is usually the common choice as it can support hybrid persistency programming. The `IPersistable` allows the user to choose which data in the object is to be serialized and which constructor to use for the deserialization operation. The `IPropertyPersistable` can only use the default inbuilt constructor for a .NET object, which is why it does not support hybrid persistency and is therefore not the recommended interface. When inheriting the `IPersistable` interface the program will inherit the `Serialize` function as shown in Listing 3.6. In this function, the information that is desired to be persistable for the object is added to `statemap` from the object store. The object information added to the `statemap` is set to a designated key, like a normal map or dictionary workflow. The storage engine internally references different graph objects for each object stored. Therefore, a key in the `statemap` can have the same value for multiple objects because the objects are treated as different graph objects in the storage engine. Meaning a developer does not need to worry about duplicate keys over different objects.

However, the storage engine cannot store all types of data. The storage engine can handle

3.3 Cleipnir Persistent Programming

the basic data types like `int`, `string`, `boolean`, etc. Unfortunately, the storage engine does not support inbuilt data structures like arrays, dictionaries, etc. The storage engine also not compatible with any newly created objects or data types outside of the basic ones. To make a custom object be serializable for Cleipnir, they need to inherit the `IPersistable` interface and have assigned their serializer and deserializer functions properly. This means data types like `enum` are not supported. However, Cleipnir supports inbuilt versions of common data structures like arrays, dictionaries, and lists that the storage engine can in fact persist. Therefore, an easy workaround is to substitute typical data structures for the inbuilt Cleipnir versions of the data structure. For instance, a dictionary object can be substituted for Cleipnir's `Cdictionary` object. For objects with a data type that Cleipnir does not support, a common workaround is to type cast it into another format that Cleipnir can persist. An example of this can be seen in Listing 3.6 where the object `PhaseType` is of `enum` type, and Cleipnir cannot persist `enum` type objects. Therefore, it is type cast to `int` while stored in memory. Then, in the deserialize process, the correct `enum` type can be chosen based on the stored `int` value. For the deserializing process, a private static function called `Deserialize` is needed, which uses the state map as a parameter. Even if the content of the function must be unique for each object's constructor, the format of the function follows the same structure shown in Listing 3.6. The `deserialize` function initializes the object through a constructor and then returns the new instance of the specified object based on the information currently stored in the `statemap`.

```
1 public class PhaseMessage : IPersistable %inherit interface
2
3 %Constructor to Deserialize process
4 public PhaseMessage(int id, int seq, int view, byte[] dig, PMessageType phase, byte[]
   sign)
5 {
6     ServID = id;
7     SeqNr = seq;
8     ViewNr = view;
9     Digest = dig;
10    PhaseType = phase;
11    Signature = sign;
12 }
```

Listing 3.5: Object persistency initializer

3.3 Cleipnir Persistent Programming

```
1 public void Serialize(StateMap stateToSerialize, SerializationHelper helper)
2 {
3     stateToSerialize.Set(nameof(ServID), ServID);
4     stateToSerialize.Set(nameof(SeqNr), SeqNr);
5     stateToSerialize.Set(nameof(ViewNr), ViewNr);
6     stateToSerialize.Set(nameof(Digest), Serializer.SerializeHash(Digest));
7     stateToSerialize.Set(nameof(PhaseType), (int)PhaseType);
8     stateToSerialize.Set(nameof(Signature), Serializer.SerializeHash(Signature));
9 }
10
11 private static PhaseMessage Deserialize(ReadOnlyDictionary<string, object> sd)
12 {
13     return new PhaseMessage(
14         sd.Get<int>(nameof(ServID)),
15         sd.Get<int>(nameof(SeqNr)),
16         sd.Get<int>(nameof(ViewNr)),
17         Deserializer.DeserializeHash(sd.Get<string>(nameof(Digest))),
18         Enums.ToEnumPMessageType(sd.Get<int>(nameof(PhaseType))),
19         Deserializer.DeserializeHash(sd.Get<string>(nameof(Signature)))
20     );
21 }
```

Listing 3.6: Serialize/Deserialize code example

Finally, Cleipnir's *CTask* class needs to be introduced. As the name suggests *CTask* shares similar traits with the *Task* object mentioned in Section 2.1. An asynchronous function that returns a *CTask* is an asynchronous operation that is to be run by the Cleipnir execution engine. In a sense using *CTask* for an asynchronous function means the operation performed inside the asynchronous function is meant to be persistable. For an object to be persisted during execution it needs to be run synchronously or in an asynchronous *CTask* operation. An example of this would be if the user wanted to persist one of the reactive Cleipnir *Source* objects, then the function waiting for emitted items need to return a *CTask* rather than a *Task*. Otherwise, the Cleipnir storage engine will crash upon attempting to persist it.

Keep in mind *CTask* should not be used when the asynchronous function has any asynchronous operations unless you intend to use Cleipnir to persist the data. Using asynchronous operations inside a *CTask* causes Cleipnir to create a new thread to handle the asynchronous operation while continuing with the rest of the operations inside the function. This also applies when the `await` operator is used for a traditional asynchronous operation, meaning the `await` becomes redundant and will not work as intended in this case. This also applies when scheduling new operations for Cleipnir inside a *CTask* function since the schedule function for the Cleipnir execution engine is treated as an asynchronous operation. A user of Cleipnir must avoid creating potential race conditions within their implementation due to running traditional asynchronous operations inside a *CTask* function. Normally it is best to try and avoid this situation entirely, thereby restricting a *CTask* function to only operate with synchronous operations. In contrast, any asynchronous operations required should be performed inside other *Task* operators using the TAP workflow discussed in Section 2.1. In short, although it is possible to call asynchronous *Task* inside *CTask* functions, the scheduler runs this operation separately in another thread. Therefore, it is recommended to keep asynchronous *Task* away from *CTask*. Instead, *CTask* should only use the TAP workflow when working with other asynchronous operations from Cleipnir, which is together with other *CTask* or when listening for events in *Source* objects. An example of a *CTask* function can be seen in Listing 3.7,

3.3 Cleipnir Persistent Programming

where we have to use `CTask` to listen for new items in the *shutemit* `Source` object which is to be persisted. The code here would result in an error in the Cleipnir object store if the `CTask` assignment were instead a traditional `Task`.

```
1     public async CTask<bool> ListenForShutdown(Source<bool> shutemit)
2     {
3         var shut = await shutemit.Next();
4         Console.WriteLine("View Change Received Shutdown");
5         return shut;
6     }
```

Listing 3.7: Example of a `CTask` function

Chapter 4

Practical Byzantine Fault Tolerance

This chapter presents the Practical Byzantine Fault Tolerance consensus algorithm in detail. We start by first introducing the system model commonly used for the PBFT algorithm. Then, a detailed explanation is given for how the protocol normally operates. This includes mechanisms such as checkpoint and leader changes.

4.1 Introducing Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance is a consensus algorithm specifically designed to handle Byzantine faults in an asynchronous distributed network. The algorithm was first published in 1999 by Miguel Castro and Barbara Liskov [30]. Notably, the Linux foundation's open-source blockchain named Hyperledger [31]–[33] uses PBFT.

The problems derived from byzantine faults originally came to light through a well-known problem known as the Byzantine Generals Problem [34], [33], [35], [36, p. 382], [37, p. 240-253]. The Byzantine Generals Problem can be summarized as a couple of army generals who are each leading their own armies, and they need to reach a decision together. The most common scenario used is that the armies try to coordinate an attack on a surrounded city. The armies can only survive if the majority of the generals agree to attack the city together or the majority agree to retreat to fight another day. There are also traitor generals that actively attempt to sabotage the order. The decision is also irreversible regardless of the action performed by the other armies. A Byzantine Fault Tolerant system is a system that can handle the issue introduced by the byzantine generals' problem and is the main goal for consensus algorithms to achieve this state. This includes the PBFT algorithm [34], [33], [36, p. 382-384].

The PBFT algorithm focuses on creating a state machine network that can withstand Byzantine failures [2, p. 456]. The protocol achieves this by providing the network with two main properties. These properties are referred to as safety and liveness. To summarize these properties:

Safety is the property that ensures that the total ordering of requests is equal for all the non-faulty participating servers. In other words, the system state should be similar to a synchronous system, operating one operation at a time, even though the system is operated over multiple remote machines.

Liveness is the property that ensures that the correct result is eventually agreed upon and returned by the system [35], [2, p. 456], [4], [30, p. 2], [32], [38, p. 403], [37, p. 257].

4.2 System Model

The PBFT consensus algorithm is implemented using R number of servers referred to as *replicas*. When a replica is down or behaving maliciously, then we say that the replica is faulty. The number of faulty replicas is represented as f . Quorum is a term used to refer to the limit of messages required to verify that the replicas in the system agreed upon a decision [38, p. 408-409], [39, p. 2]. A single replica is chosen as the leader called primary and is represented as p . The other replicas are referred to as backups. The responsibility of the primary is to order the request sent to the system by numerous clients [2, p. 456], [38, p. 405]. The replica chosen as the primary replica for the PBFT network is based on the replica's identifier value [37, p. 258].

According to [30, p. 3], [38, p. 405], replicas in the distributed network move through "successions of configurations known as views." A simpler definition for a view is the number that defines the set of non-faulty replicas which are participating in the current PBFT protocol round set up by the current primary. The current view number is denoted by the letter v . As previously mentioned, the primary is chosen based on an identifier value i . That identifier value is determined by the formula $p = v \text{ mod } R$ [32], [37, p. 258], [30, p. 3]. We decided to set the initial view number to zero, which results in the formula setting replica zero as the initial primary.

The protocol can only guarantee the safety and liveness properties of a system if the number of faulty replicas does not exceed a specified margin of the total replicas in the network. The total number of replicas required to be in the system should be derived by the formula $R > 3f + 1$. From the formula, it can be determined that for each new faulty replica participating in the PBFT network, three additional replicas are required to keep the safety and liveness properties for the PBFT network. As an example, the lowest number of replicas a system can have is four. In this situation, the system can only handle up to one faulty replica. In order to handle more faulty replicas, the system has to scale up by adding three additional servers for each faulty server that exists in the system [32], [35], [37, p. 257], [38, p. 403], [30, p. 3].

All the messages sent between replicas are expected to be digitally signed by their sender. The signature process uses public-key cryptography [37, p. 257, p.267]. A hidden private key is used to sign the messages, while the other parties can use the replica's public key to verify this signature [38, p. 417]. The signature procedure is used to verify that the sender is whom they claim to be [30, p. 3]. In some cases, the digital signatures are replaced with a Message Authentication Code (MAC). This is done to remove potential

bottlenecks in performance and to detect tampering in messages [40], [37, p. 257], [30, p. 3, 8]. In this PBFT implementation, digital signatures are used for all message types.

4.3 Detailed Protocol Operations

The PBFT consensus protocol is divided into three phases. The Pre-Prepare, Prepare, and the Commit phase. If the PBFT protocol operations are properly executed, a consensus has been achieved for an operation once all three phases have transpired on $2f + 1$ replicas [35], [37, p. 257-259]. The roles of the pre-prepare phase and prepare phase are to propose an ordering for requests delivered to the system. On the other hand, the combination of prepare phase and the commit phase establishes the execution order for the replicas in the system [30, p. 4]. Figure 4.3.1 shows an illustration of the PBFT workflow. The illustration shows the messages sent from the different replicas during the different protocol phases in PBFT.

The PBFT protocol starts once a client sends a request containing their desired operation to the primary [30, p. 4]. Sometimes the client also multicasts their request to the other replicas in the system, which is the model that we followed in our implementation [39, p. 2], [38, p. 406], [37, p. 258]. Regardless of which of these models is used for the requested message, the primary is the replica responsible for starting the iteration of the PBFT algorithm to process the client's request. The primary creates a Pre-Prepare message and assigns the request with a sequence number which is then multicasted to the other replicas in the network with the same view number as the primary. Once a replica receives the Pre-Prepare message, it validates the Pre-Prepare message. The validation process consists of the following [32], [30, p. 4], [37, p. 259]

- Validating the Signature in the message.
- Checking that the view number in the message matches the current view number.
- The message sequence number is not out of bounds of the current sequence number interval [32], [30, p. 4].
- Make sure the replica has not already received another Pre-Prepare message with the same sequence number but with a different request.

Once the validation process is finished, the replica officially starts the prepare phase by creating a prepare message and multicasting it over the network. The prepare phase ends for a replica once it has stored up to $2f + 1$ validated pre-prepare/prepare messages from different replicas. After this condition is met, the replica enters the state known as *prepared*. In this state, the replica logs the message data thus far in what is called a *prepare certificate*. A prepare certificate is essentially a record showing that the prepared phase is finished and properly executed for that given request. The proof provided in a prepare certificate is a list of the valid prepare messages, basically confirming that quorum has been reached for the certificate when the number of messages stored in the

4.3 Detailed Protocol Operations

list is higher than the desired limit of $2f + 1$ [38, p. 408], [2, p. 457]. The last phase is the commit phase, which functions very similarly to the prepare phase. Each replica that is finished with the prepare phase starts the commit phase by multicasting commit messages to the other replicas in the system [30, p. 4]. In this phase, the primary functions exactly the same as every other replica. The validation process is also the same as it was for prepare messages. The goal for the commit phase is also the same as in the prepare phase, which is for a replica to receive $2f + 1$ commit messages, which includes the replica's own commit message [30, p. 5]. Once a replica has received enough commit messages, the protocol reaches the *committed* phase for the replica. This essentially means that a commit certificate is created and logged similarly to a prepare certificate [38, p. 409], [2, p. 457]. When a replica has finished both a prepare certificate and a commit certificate, then consensus has been achieved, and each replica performs the operation requested by the client [38, p. 409], [30, p. 5]. After the operation is executed, each replica sends back a reply message containing the appropriate identification values and the result of processing the given request. The last requests sent by the clients are also stored in memory to account for the situation where the client does not receive the reply messages. In this case, the client will re-transmit the same request to the system, and the replicas will re-transmit their reply for that following request [38, p. 409]. A client will accept the result if it gets $f + 1$ replies back from the replicas.

The replicas can only handle a certain amount of requests before the system is required to save its state. As mentioned in the validation process, a replica can only process new protocol messages as long as the replica can exhaust a sequence number within a given sequence number interval. Once the replica no longer can exhaust any sequence number within the sequence number interval, the replica can no longer process incoming requests until the interval is updated. This sequence interval length is always constant and is adjusted based on the systems checkpoint period, which is discussed in the next section, Section 4.4 [37, p. 262], [30, p. 4-5].

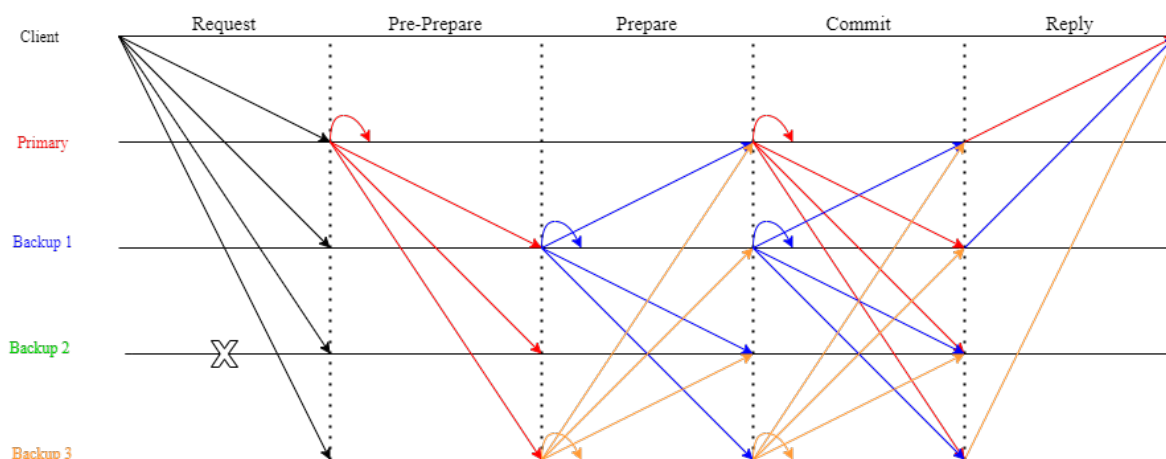


Figure 4.3.1: Practical Byzantine Fault Tolerance Normal Workflow

4.4 Checkpointing

PBFT also incorporates checkpointing, which is a mechanism used for garbage collecting the logs. Checkpointing is required so that the replica does not use up all of its memory for logging messages [37, p. 261]. Therefore, the replicas must agree upon a point where the system is stable for all the replicas. Afterwards, the replicas can delete any records in the logs prior to the consented state [30, p. 5], [38, p. 410].

Checkpoints are essentially the state records of the system after progressing a specific interval of requests. The checkpoint has information regarding the last sequence number that was performed for the system. This sequence number is used on the garbage collector to put an upper bound on the records that are to be removed. For instance, if the stable sequence number was set to 50, then the garbage collector would remove a set of logged data up to 50. The checkpoint also has a digest of the system for that stable sequence number. This digest is used to confirm that the replicas have the same system state for the given sequence number [30, p. 5], [38, p. 410].

For replicas to validate checkpoints, they each must multicast a checkpoint message over the network containing the information mentioned above together with the replica id. Like the rest of the PBFT protocol messages, a checkpoint is considered to be valid for a replica if it has stored $2f + 1$ checkpoint messages with different replica ids with the same stable sequence number and system digest [37, p. 261-262], [30, p. 5], [38, p. 410]. Once a checkpoint has been validated successfully, it is referred to as a stable checkpoint [39, p. 3], [37, p. 261]. The replica usually stores checkpoint messages for different sequence numbers in memory and has only a single record for a stable checkpoint. Once a new stable checkpoint is determined, any checkpoint records with lower sequence numbers are removed from memory. If there exists a previous stable checkpoint in memory with a lower sequence number, then it is replaced by the new one [37, p. 261-262].

In PBFT, checkpointing is usually performed periodically after a constant number of requests have been processed. This interval length is constant and is referred to as a checkpoint period [37, p. 261], [38, p. 410]. As mentioned earlier, in Section 4.3, PBFT normally only processes a sequence number in the set of currently available sequence numbers. The length of the sequence number interval is designed to follow the format $[checkpointinterval + 1 - 2 * checkpointinterval]$. This means the system attempts to calculate two checkpoints during a single sequence number interval. Once a stable checkpoint is obtained, the system extends the sequence number interval where the new interval starts at the last stable sequence for the current stable checkpoint [30, p. 5], [38, p. 410]. Unless a replica has exceeded the upper bound of the sequence number interval, the replica usually performs the checkpoint functionality concurrently with the protocol workflow.

4.5 View-change

In the scenario in which the primary is the faulty replica, a view-change eventually occurs. The purpose of the view-change is to reassign the responsibility for a primary away from the current primary replica that is deemed faulty, which is then given to another replica that is not faulty [41], [37, p.262]. As mentioned in Section 4.2, the replica that is chosen as the next primary is based on the replica id and the next view number. Therefore, the view-change updates the view number for the system to change the system's primary replica. Some operations have to be performed for a view-change process to be deemed successful. The first operation is to update the view number to set another replica as the primary [30, p. 6], [38, p. 411], [42]. This step includes multicasting view-change messages between replicas to start the new view session. The other more demanding operation is that the primary needs to make sure that the system is stable and that replicas start the new view with the exact same system state. Therefore, all requests performed after the last stable sequence number need to be reprocessed between the replicas. This is done so that the system can guarantee that the replicas are not missing any of the previous operations performed to the system. [2, p. 458], [37, p. 263-265].

There are several ways for a replica to deem its primary to be faulty. The most common way is to have a timeout functionality for the protocol execution. It is most common to start a timeout once a replica has received a request from the client. Suppose the replica does not accept any pre-prepare messages for that request before the timeout expires. In that case, the replica goes into view-change mode and no longer participates in any of the protocol operations [32], [30, p. 5-6], [37, p. 263].

The view-change process starts by having the replica increment its view number. Then the replica creates, signs and multicasts a view-change message over the network. The replica then waits for $2f + 1$ view-change messages [32], [30, p. 6], [38, p. 411], [42]. A timeout is also used here. If the replica does not receive enough view-change messages in time, the process repeats with the next incremental view number. In some cases, a replica can also be designed to go into view-change mode if a replica has already received two view-change messages from other replicas, as it now only requires its own view-change message for the system to agree that a view-change is necessary [37]. Once the appropriate number of view-change messages are received, the new primary is responsible for creating, signing, and multicasting a new-view message to the other replicas [37, p. 264]. Before the new-view message can be multicast to the other replicas, a new primary must go through its log and each of the protocol certificates received from the view-change messages. This process is done so that the new primary can create new pre-prepare messages for all sequence numbers that have occurred after the last stable sequence number. If the new primary lacks information for any of the sequence numbers, the new pre-prepare message has its request digest set to the value null. This information is included in the new-view message, which is then sent to the other backup replicas. The backup replicas then validate and reprocess each of the sequence numbers that have a valid pre-prepare message. This essentially means that the other replicas have to multicast a new prepare message and then participate in a commit phase together with the new primary for each of the pre-prepare messages in the new-view message [30, p. 6], [2, p. 458], [37, p. 265]. A timeout is once again being used to handle the situation where the reprocessing takes

4.5 View-change

too long. This process can also fail if the pre-prepares in the new-view message fails the validation process. If either the timeout occurs or the validation fails, it is back to the start of the view-change process. Once all pre-prepares have been reprocessed, the view-change procedure is over, and the replica returns to normal protocol operations with the new chosen primary. Keep in mind that any new requests received during the view-change process are ignored by the system [37, p. 263].

Figure 4.5.1 shows an example of a view-change process. The figure shows the timeline for each of the processes needed for the view-change to be successfully completed. Starting with the timeout occurring on the backup replicas when the primary is no longer working correctly. Then the replicas each multicast a view-change message to the other replicas in the system, including the faulty primary. After the new primary has received a sufficient number of view-change messages, it creates pre-prepares messages that need to be reprocessed in the network. Afterwards, the new primary multicast new-view messages to the other replicas to start the reprocessing phase. Finally, the system multicasts both prepare and commit messages to validate pre-prepare messages. The system then moves on to normal workflow, with the first backup replica now serving as the primary for the system.

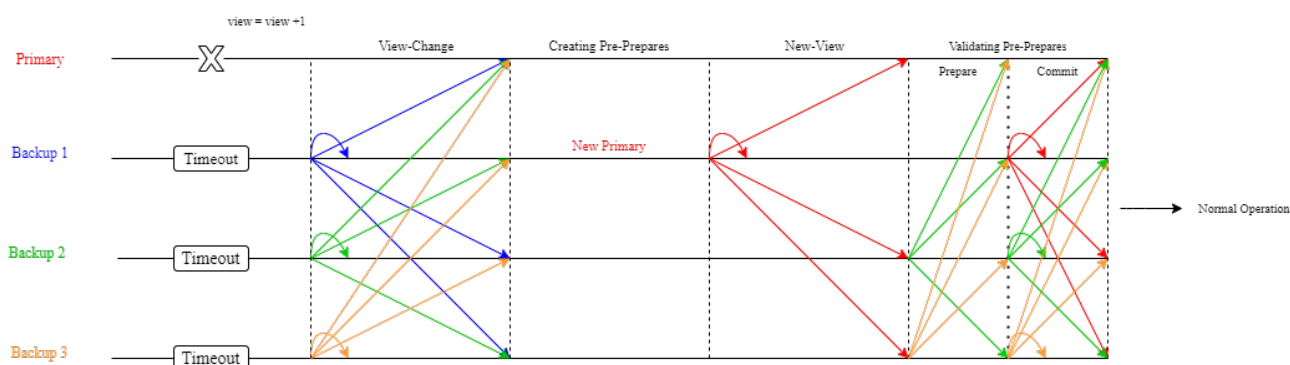


Figure 4.5.1: Practical Byzantine Fault Tolerance View-Change

Chapter 5

Related Work

The University of Stavanger has previously supported the development of Cleipnir. Therefore, there exist previous papers and thesis on Cleipnir usage for implementing consensus algorithms. Two contributions, in particular, have been used as building blocks for this project. We now discuss these two works in detail and explain how they contributed to our work.

5.1 Cleipnir - Framework Support for Fault-tolerant Distributed Systems

This paper is the original paper describing the Cleipnir framework. It was written by its creator Thomas Stidsborg Sylvest with the help of two professors at the University of Stavanger, Leander Jehl and Hein Meling. The paper describes, in detail, the internal functionality and tools available in the Cleipnir framework. The paper describes Cleipnir's use cases and why Cleipnir prioritizes these functionalities. The paper has detailed explanations for how the tools work together with practical demonstrations. The demonstrations are presented using an existing implementation of the Paxos consensus algorithm. The paper also presents a Raft implementation using the Cleipnir framework. This includes the overall architecture of the implementation and detailed examples of how Cleipnir is used to simplify tasks performed in the Raft algorithm. Finally, experiments are performed to evaluate the performance of the Raft implementation. The results of the experiments are compared directly to an earlier Paxos implementation. The evaluation performed focuses both on latency and code structure. Our thesis is a direct continuation of this paper with relatively similar goals. The largest difference between our thesis and this paper is the chosen consensus algorithm to be implemented using Cleipnir. Additionally, we do not evaluate our PBFT implementation in terms of latency. Our contribution is to provide additional experiences on how well Cleipnir can be utilized in implementing complex consensus algorithms. This also implies discovering potentially difficult problems that the current Cleipnir framework cannot handle, specifically, whether or not Cleipnir can handle all of the complex issues within the PBFT algorithm while still having a

simple-to-read code structure [5].

5.2 Implementing a Distributed Key-Value Store Using Corums

In 2010, Eivind Bakkevig wrote a master thesis about Corums. In his thesis, Bakkevig used a .NET framework called Corums to implement a dictionary-based distributed system. This Corums based implementation implemented the Paxos consensus algorithm to make decisions for the dictionary-based distributed system.

The Corums framework is the predecessor to the Cleipnir framework. It follows the same programming models as Cleipnir does. These models would be the ones described in Chapter 3; built-in persistency, reactive programming, and a single-threaded scheduler. The main difference between Corums and Cleipnir is that Corums focus more on simplifying abstraction for developers to handle communication using incoming/outgoing communication buses. Cleipnir instead focuses more on giving the developer the tools necessary to develop consensus algorithms that follow the persistent program paradigm in an easy-to-use and customizable manner. As an example, a major difference between Cleipnir and Corums frameworks lies in Corums support in reliable message delivery between distributed systems. Corums has support for bus abstraction that can simplify the process of handling incoming/outgoing messages between the nodes in the system. Cleipnir does unfortunately not support this functionality. Instead, Cleipnir prioritized evolving the persistence functionality previously provided by Corums [5, p. 6-7], [6].

Corums is very similar to Gorums [43], [3, p. 22], which is intended based on how close the names are, the main difference being the supporting language. Bakkevig succeeded in creating a distributed dictionary storage using the Corums framework. Additionally, Bakkevig built the client-side for the implementation using ASP.NET Core Web API [44].

According to Bakkevig, he had no prior experience with the C# language before writing his thesis. Bakkevig did, however, have previous experience with the Paxos consensus algorithm. This made most of his work during the thesis about learning C# and the Corums framework rather than extensively researching Paxos. As for our thesis, the exact opposite is true. We have some background knowledge regarding the C# language but had little to no background knowledge of the PBFT algorithm. Therefore, much work for this thesis revolved around learning and making our own PBFT algorithm based on its description. Although we had experience with the C# language, we had no previous experience with the Cleipnir framework. Therefore, similarly to Bakkevig, our thesis also required us to study the Cleipnir framework. [3, p. 8].

Chapter 6

Design

This chapter presents an overall summary of our PBFT application implementation. This includes a description of the system architecture used for the PBFT network. Additionally, to better understand the application's workflow, a summary of our code structure is given. Finally, a description is provided for the current application design. Primarily specifying which parts of the program take advantage of the Cleipnir framework and how the server-side interacts with the protocol workflow.

6.1 Network Architecture

Figure 6.1.1 shows the system architecture used for PBFT implementation. Generally, our network architecture follows the same structure as the system model introduced in Section 4.2. The system consists of four server implementations called *replicas*, where the replica with the lowest identifier value is chosen as the primary. These four replicas are communicating over a mesh network using socket connections. This means that each replica shares a unique network socket with each of the other replicas in the PBFT network. To avoid creating multiple socket connections between two replicas, the replica with the highest identifier is tasked with being the initiator when creating the socket connection. Because of this, the primary replica is not required to actively establish any of its connections to its fellow replicas. Instead, the primary establishes all its socket connections by listening for any connection attempts on its local network address. The opposite scenario occurs for the replica with the highest identifier value. Although the other non-primary replicas also listen for connections on their local network address, this is done to connect to replicas with higher ids in the network. The non-primary replica is also responsible for initiating the socket connections with all the other replicas in the network with lower identifier values.

When the replicas have established connections, the replicas can still not fully communicate until they have exchanged public keys. This is required so that messages between replicas can be verified using a digital signature. Public keys are exchanged in *session messages*, which are messages that are automatically sent between replicas once a socket

6.1 Network Architecture

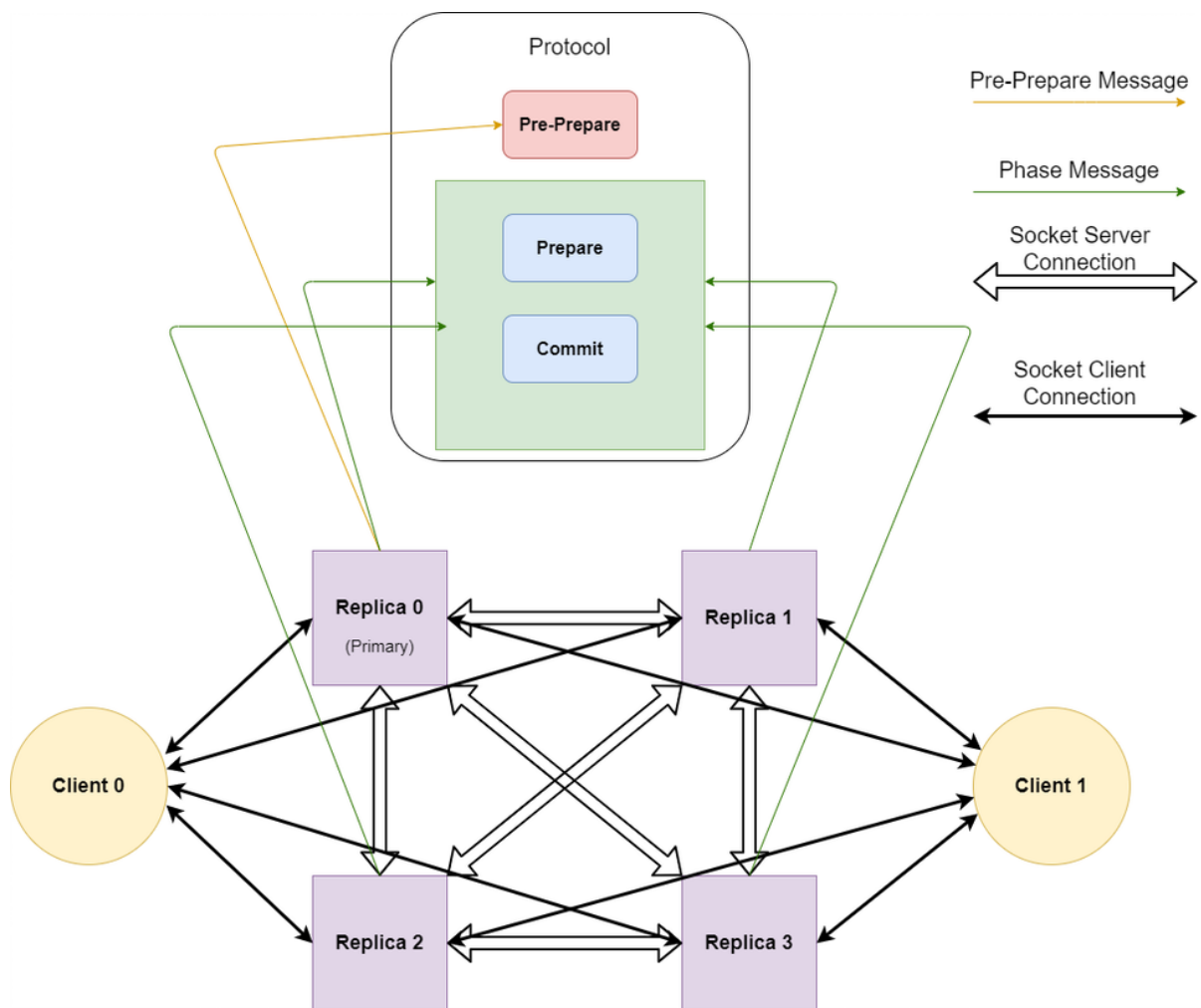


Figure 6.1.1: Overall architecture of the PBFT implementation network

connection has been established. If the public keys are for some reason not exchanged, then the replica discards any message received from that host. If the message received from the unknown host happens to be a *request* or a protocol-related message, the replica also terminates the connection. This also applies to clients. This current public key model is unfortunately not very secure. This is due to public keys being ephemeral. Therefore, it needs to be updated and replaced if the replica's execution is interrupted or stopped. In this implementation, the private and public key pair for a replica is created at the system start-up. Currently, there is no way for the replicas to authenticate another replica after it has rebooted. Therefore, a replica must replace the key-value pair representing a replica's public key in its register if it receives a new session message with the equal replica identifier value. This, in turn, means the system is susceptible to impersonation and spoofing attacks [45]. Because the main goal of this thesis focuses more on the aspects behind implementing a simple PBFT workflow, the current cryptographic system was deemed sufficient for simulating a network using digital signatures. However, it is important to be aware of this major security flaw of the system to be potentially fixed and avoided in the future.

6.2 Overview of Workflow

The system performs the PBFT protocol by exchanging protocol messages over the mesh network until at least three replica implementations have finished all three protocol phases. The PBFT protocol is triggered when the server receives a request from one of the connected client nodes. The primary is responsible for officially starting an instance of the PBFT protocol by multicasting a protocol message of type pre-prepare. There are two important goals for the pre-prepare phase. The first is to make sure that the replicas have an agreement upon the ordering of the request. In other words, the replicas perform the requests in the same order as the primary, which means a request processed over several replicas should all be using the same sequence number for processing that request. The second important goal is to determine whether or not the primary is fit to be the leader. As mentioned in section Section 4.5, a view-change occurs when a leader no longer is eligible. The view-changes are triggered by timeout functionality in our application, which is set once a replica receives a client request. If the primary takes too long in the pre-prepare phase, then the timeout occurs, and the other replicas perform a view-change to change the primary replica. The rest of the replicas are the responsible parties during the prepare phase by sending protocol messages of type prepare, while every replica participates in the commit phase using commit type protocol messages. The last step of the current PBFT implementation is to create a reply message and send it back to the client responsible for the request. The details in regards to the PBFT workflow implementation is discussed in Chapter 7.

6.3 Code structure

In Figure 6.3.1 shows a figure that illustrates a short summary of the code structure used for our PBFT replica implementation. The summary shows folders containing our source code used to implement the necessary items for the PBFT algorithm. The summary also highlights some of the more important files, meaning files that contain the most relevant code segments for the application.

6.3.1 Protocol Objects

To start off, the PBFT algorithm uses many different types of messages for the replicas to collaborate. For simplicity, we stored all of the classes revolving messages within the *Message* folder. Since the protocol messages share traits and functionality, we introduced two interfaces to reduce redundancy. The first interface regards the serialization process for transforming the messages into byte streams so that to allow the message objects to be exchanged over the Transmission Control Protocol (TCP) [46]. The other interface is for the digital signature process. Session messages are not signed; therefore, they do not inherit this interface. In addition, to reduce complexity, we set the normal workflow protocol messages such as pre-prepare, prepare and commit messages, to use a single class object known as **Phase Messages**. A single value within the **Phase Message** object designates which protocol type the **Phase Message** represents. View-change and checkpoint messages are instead represented with their own unique class object.

To store the proof of an PBFT iteration, we implemented class objects known as *certificates* found in the *Certificates* folder. The implemented certificate objects essentially have the same functionality as the different certificates described in Chapter 4 for the PBFT algorithm. Certificates act as records showing the state of protocol iteration, where a single iteration requires two valid certificates before the iteration is deemed finished. Just like the protocol messages, we implement both protocol certificate types by an object class known as **Protocol certificate** where a single variable determines the protocol types. Checkpoints and view-changes once again have their own class object for their certificate proof. There are also interfaces used to avoid redundant code for the certificate objects. The interfaces used for certificates are primarily used to add the mandatory functions for a given certificate object.

6.3.2 Other functionalities

The static functionalities that are not tied to any object classes are placed in the *Helper* folder. This includes functionality for serializing and deserializing messages objects with JavaScript Object Notation (JSON) [47], cryptographic functions related to creating and validating digital signatures, and files containing all the enum types used for this implementation. An enum is essentially a predefined .NET class with only constant values. The enum's constant values are defined upon initialization and are helpful for classifying

6.3 Code structure

other objects [48]. For instance, our PBFT implementation has used enums to categorize the protocol phase a phase message belongs to. This allows the program to easily distinguish between pre-prepare, prepare and commit phase messages even when they all use the same object type.

The *JSONFiles* folder contains the JSON files, which have information about the network addresses for the replicas in the system designated to their receptive identifier values. There exist two JSON files in this folder. The first file is used when running the implementation over multiple systems or over docker containers. The second file uses localhost addresses with different port numbers that are meant to be used when testing the application on a local device.

The PBFT replica implementation uses Cleipnir to persist important parts of the code for servers to be able to reconnect to the system easily. As discussed in Chapter 3, Cleipnir has several different engine types, which can be used to serialize and store the application's data. In our PBFT replica implementation, we have decided to use the *Simple File Storage* engine. The *Storage* folder will contain the .txt files in which Cleipnir stores its data. Since there are several instances of replicas in the PBFT network, the name of the .txt file used to store the application data will follow the structure "PBFTStorage" with the replicas identifier value at the end of the name.

The *Replica* folder contains code that is directly related to the server or the protocol workflow. The *Replica* folder has two subfolders in order to distinguish code based on their functionality easier. All the networking code that is not directly connected to the server-side network handling is placed inside the subfolder *Network*. This includes the code for creating sockets and functions for properly handling the data received from the socket by the TCP network protocol. The *Protocol* subfolder contains code that is directly related to the protocol execution. This includes code related to the execution of the main workflow of the PBFT algorithm. In addition, code for handling the reactive execution for view-change and checkpointing is also placed here. The other files in the *Replica* folder contain code that helps the replica run properly, including code used to help communication between server-side and protocol workflow run inside Cleipnir.

6.3.3 JSON Serialization Problem

As mentioned in Section 3.3 Cleipnir requires a designated serializer and deserializer function to persist an object. In addition, Cleipnir cannot persist traditional data structures directly. Instead, Cleipnir intends for the developer to substitute traditional data structures with Cleipnir respective inbuilt data structures. This means that whenever Cleipnir persists objects that have a reference to a data structure, then the persisted object must also substitute the data structure for the correct inbuilt Cleipnir data structure. For our implementation, this is present in our `Certificate` objects as they all require a list of proofs. Therefore to persist the proof list, we need to replace a traditional `List` class object with an inbuilt `CList` class object. However, Cleipnir inbuilt data structures cannot be serialized or deserialized properly using JSON formatting. As we were not completely aware of this issue when we began designing our application, we, unfortunately, made an

6.3 Code structure

oversight for our network layer. The application currently uses JSON [47] to serialize and deserialize messages when sent over the PBFT network. Since JSON formatting does not support inbuilt Cleipnir data structures, which in retrospect was not all that surprising, attempting to deserialize any protocol message that includes an inbuilt Cleipnir data structure, causes the application to crash. Currently, we solve this issue by converting between traditional data structures and inbuilt Cleipnir data structures whenever a message with said data structure is to be serialized with JSON. The conversion itself is rather primitive. A copy of the data storage in the traditional data structure format is created, then the content is copied from the source data storage to the newly created data storage. The process is then reversed on the receiver end after the message has been properly deserialized. To further simplify the conversion between the data structure types, we added temporary JSON class objects that act as substitutes for the persisted objects that have issues with JSON when they are to be sent over the PBFT network. Transforming between a persisted object to their respective temporary JSON object is as simple as making a function call with the opposite transformation also available as a function tied to the temporary object. These JSON objects are all placed in the subfolder `JsonObject`s inside the `Helper` folder. All implemented JSON class objects follow the naming convention *JSON + Objectname*. Having now learned about this issue, it would have probably been more beneficial if the serialization and deserialization for the networking used the same formatting that Cleipnir supports to avoid this issue in future projects.

6.3.4 Notable Files

The *App.cs* file contains the code which starts all the processes needed in the PBFT replica. This includes code for starting Cleipnir, creating a server instance, and starting the protocol handlers. This file also includes the main application state, which in this implementation is simply a list of operations theoretically performed by the system.

The *Server.cs* file is by far the largest, and this is due to the server implementation acts as the bridge between the network layer and the protocol workflow. This is necessary as the network layer is responsible for formatting protocol messages being sent to and received from the PBFT network. On the other hand, the protocol workflows require the protocol messages to finish all of their jobs. The protocol workflows receive the messages from the network layer by having the server implementation schedule emits to the respective workflows `Source` object. The server is also responsible for the other server-side operations.

The *Workflow.cs* file is where the code for normal protocol workflow and view-change workflow occurs. Chapter 7 introduces in detail how the workflows are implemented.

6.3 Code structure

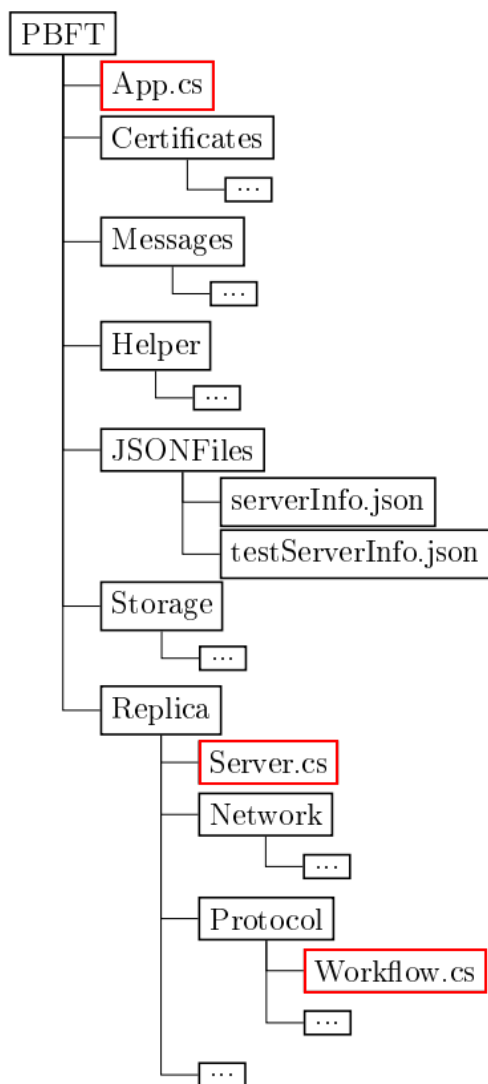


Figure 6.3.1: Summary of the file architecture for the PBFT implementation

6.4 Persistent vs Ephemeral

An important detail when using the Cleipnir framework is to have a general idea for which parts of the system are desired to be persistent. As mentioned in Section 3.3, Cleipnir allows for hybrid persistent programming allowing the developer the freedom to choose which parts of the application to persist while keeping the rest of the data ephemeral. In our case, it was essential to figure out which data is needed to be persisted for the PBFT algorithm to be easily reinstated should the system shutdown. In addition, when taking advantage of hybrid persistent programming, it is important to avoid storing a lot of unnecessary data as it generally slows down the synchronization period. Another reason why making a distinct divide between persistent and ephemeral parts of the code is important is due to the difficulties encountered when using `CTask` together with traditional asynchronous operations, as mentioned in Section 3.3. In our case, because our goal is to evaluate both `async/await` and Cleipnir for implementation of consensus algorithms, it is crucial to not interchange these tools as it would, in most cases, lead to race conditions.

Figure 6.4.1 shows parts of our PBFT implementation that are divided into persistent parts and ephemeral parts. In addition, the figure shows an illustration of how ephemeral parts and persistent parts collaborate using arrows to dictate the program flow. In general, for our application, static functions and objects unrelated to the PBFT workflow are treated as ephemeral. Meanwhile, objects related to the PBFT implementations, `Source` objects and functions that handle workflow related to the PBFT protocol are persisted. The server is an exception as some parts are persistent while others parts are not. For example, the protocol logger is stored in the server and is persisted; on the other hand, any code related to networking in the server is not persisted. We can view certain operations occurring in the server as being treated as a bridge between the persistent part and the ephemeral parts. By this, we mean that the server is responsible for handling any messages received from the ephemeral socket connections and delivering them to the appropriate protocol workflow that is persistent.

These messages are then emitted to their respective reactive operators, which are run in the persistent part and are waiting for changes to occur. There are a couple of exceptions to this rule. For instance, if the message received is needed for the server-side, the message is used directly by the server. A primary example of this being session messages. However, for the server's non-persistent network handler to perform operations that affect the persistent part run by Cleipnir, the operation must be scheduled by Cleipnir's execution engine. There are two reasons for this syntax.

Firstly, code run by Cleipnir is not supposed to be affected by code outside of Cleipnir. However, since the protocol workflow relies on messages from the non-persistent network layer, our application cannot avoid it. Therefore, a way for the persistent and non-persistent areas to work together must be coordinated. To start, attempting to emit a message to a `Source` object or perform an operation that changes the state of a persisted object outside of a `CTask` function can cause the same scenario discussed earlier in Chapter 3. That is, new threads are created, and Cleipnir attempts to perform the operation concurrently. However, this scenario is not always thread-safe and sooner or later causes issues for the system state.

6.4 Persistent vs Ephemeral

It is intended to avoid this problem by using the Cleipnir execution engine directly to schedule the desired operation to be run with Cleipnir support. The Cleipnir execution engine runs its scheduled operations using a FCFS approach, as mentioned in Section 3.1. Scheduling operations desired for the protocol using the execution engine makes it easier to keep track of operations, as they most likely run in a synchronous order with the other operations happening in the protocol workflow. However, this does not mean the system is synchronous, as Cleipnir only gives an illusion of a synchronous system. The execution engine can change the order of execution while another scheduled operation is idle or stuck. In which case, the next scheduled operation is run. Listing 6.1 shows an example of how to schedule an operation to the Cleipnir execution engine. In this example, the server schedules a `Phase message` to be emitted to the `ProtocolSource Source` object.

To summarize, the relationship between the server, network layer, and the persistent protocol workflows is that the server is responsible for emitting any relevant messages received from the network layer to the protocol workflow. In addition, the server is also responsible for preparing and sending any protocol messages it gets from the protocol workflow to the network layer so that the protocol messages are correctly sent to the other replicas in the network. To accomplish this functionality, the protocol workflows have an object reference to the server. This means the protocol workflow calls the appropriate send function in the server object whenever a protocol workflow needs a protocol message to be sent over the network. Since both the normal protocol workflow and view-change workflow are required to multicast protocol messages to move on in their respective workflows, the workflows must directly reference the server object's multicast function. All protocol workflows that require the server reference are all a part of the class known as `Workflow`

This design is unfortunately not perfect. Ultimately, since the server is responsible for scheduling the operations revolving around protocol messages, the protocol workflows can become stuck if something happens to the scheduled emit. We have encountered some situations where the scheduled operations to the Cleipnir execution engine never finishes its execution. This would not be considered a big issue most of the time as the application is run asynchronously. However, there are some instances where this situation can become a big problem. Usually, when scheduling an operation that requires emitting an item to a `Source` object that currently does not have any receivers, the operation never finishes. When the server attempts to schedule an additional operation afterwards, the operation is never scheduled, leading to the application getting stuck. This situation seems to be similar to sending messages to a channel without any receivers in Golang [49], despite that this is an issue that rarely occurs, but when it does, it is detrimental to the system. Therefore, to counteract this issue, strict conditions are placed in the server to avoid sending messages to reactive subjects in situations where there are no listeners.

6.4 Persistent vs Ephemeral

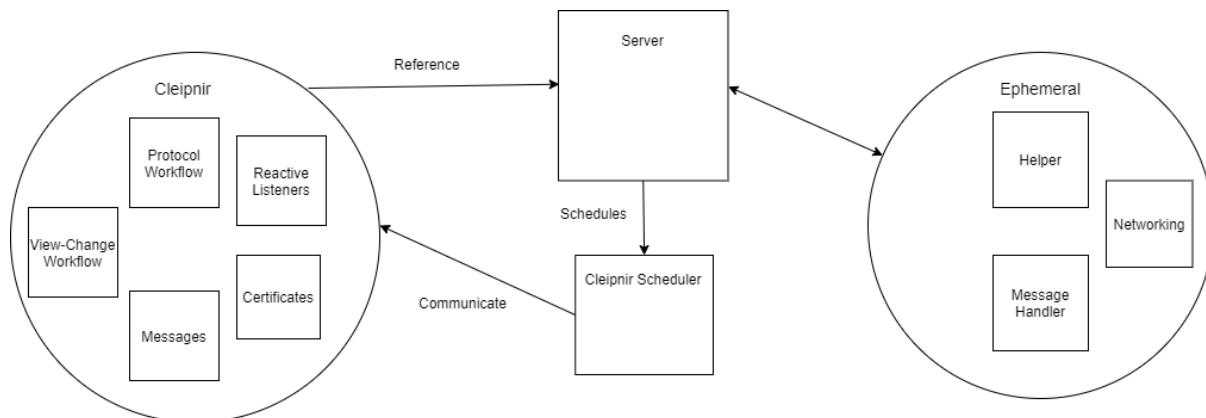


Figure 6.4.1: Application divided into persistent parts and ephemeral parts and how they interact

```
1 public void EmitPhaseMessageLocally (PhaseMessage mes)
2 {
3     Console.WriteLine("Emitting Phase Locally!");
4     if (ProtocolActive)
5     {
6         _scheduler.Schedule(() =>
7         {
8             Subjects.ProtocolSubject.Emit(mes);
9         });
10    }
11 }
```

Listing 6.1: Example of server and protocol interaction using Cleipnir scheduler

Chapter 7

Implementation

In this chapter we introduce our PBFT implementation. We will introduce the implementation for the request handler, normal protocol workflow, view-changes, and finally checkpointing. We will also discuss how the Cleipnir framework is used to create the working PBFT implementation, as well as discuss some benefits and limitations within the current implementation design.

7.1 Design Choices

With the goal of the thesis in mind, the PBFT protocol workflows were designed to be as close defined to the protocol description as possible. To accomplish this, we believed the best approach would be to design protocol-related workflow as orderly as possible, meaning we generally want to use synchronous programming workflow whenever it is possible. Several factors persuaded us to focus on keeping the majority of the protocol operations synchronous. The first reason was that it is generally easier for developers to keep track of the program's progress, making it easier to debug. Modern `async/await` workflow can generally achieve a similar program workflow to that of synchronous workflow. However, unless you are worried about blocking the main thread, there are no added benefits in terms of complexity or efficiency in using asynchronous programming over synchronous programming. The second reason is in regards to using Cleipnir for our protocol workflows. We previously discussed in Section 6.4 and Section 3.3 that using normal asynchronous operations inside a function that uses Cleipnir is not a good idea. Because we wanted to take advantage of reactive programming to handle protocol-related messages in our protocol workflow, we needed to use Cleipnir reactive framework. In addition, since persistency is a core part of the Cleipnir framework, we decided it was also best to keep persistency in mind while designing our application. Therefore we decided to keep our protocol implementations inside `CTask` functions. Because of this, the only form of asynchronous operations that are performed inside any of the protocol workflows are restricted to other `CTask` asynchronous operations. The `await` operator still works the same as it does for traditional asynchronous operations. Allowing us to take advantage of the `await` to set waiting points for `CTask` operations as well as on-

7.1 Design Choices

going reactive streams; Thereby giving the protocol workflows an abstraction to that of synchronous workflows, even though, in reality, it is an asynchronous process. To enable Cleipnir to persist in our protocol workflow, we also had to make sure that objects we wanted to persist in the protocol were persistable. To accomplish this, we initialized both a serializer and deserializer functions for Cleipnir that follows the format specified in Section 3.3 to use for each of our defined protocol objects. Finally, to take advantage of the Cleipnir persistency functionality, we have done our best to avoid creating circular dependencies. Circular dependencies would essentially cause the serialization process to fail, as it would lead to two references that depend on one another. We believe there are no circular dependencies in our current implementation because the Cleipnir serialization process does not crash during Cleipnir's synchronization process. However, if there does exist a circular dependency in our application, the server's relationship with the protocol workflows would be our primary suspect. This is because the server emits messages to the protocol workflow while the protocol workflow has an object reference to the server. We believe they do not have a circular dependency because the server interacts with the protocol workflow through the Cleipnir execution engine and does not directly reference the protocol workflow. However, if our assumption is wrong, then this design would have to be changed in the future.

To make it easier to understand the protocol workflow, we believed the best approach was to keep only the protocol processes described in the descriptions centred inside a single function or class whenever it was possible. We chose this design primarily because we wanted to make the code as readable as possible. It was deemed especially important when designing the standard PBFT workflow. This design was not entirely possible to replicate for checkpoint and view-change workflows. We also attempted to keep operations unrelated to the protocol outside the protocol workflow. Although in some cases, we cannot avoid this issue. In these cases, a simple function call with a good function name must suffice to avoid increasing the complexity of the overall workflow. An example of this is using the server to send protocol messages to PBFT. It is a fundamental part of the PBFT consensus algorithm to interchange protocol messages. However, the operations performed in the sending operation itself do not affect the protocol workflow. Therefore, a simple call to the servers `Multicast` using the newly created protocol message as a parameter should be decisive enough for readers of the workflow.

Another important topic discussed in Section 6.4 was the need to use the Cleipnir execution engine to schedule operations when operations outside of Cleipnir are required to affect persistent systems. To simplify this design in our application, we made practically all of the scheduled operations that use the Cleipnir execution engine be performed within the server. This design is chosen to make it easier to keep track of where the items are emitted to the protocol workflows. The server has several emit functions ready for scheduling the given message type to its desired `Source` object. In addition, all functionality in regards to handling and sending incoming protocol messages from the PBFT network to their respective protocol workflow is centred in its own class called `MessageHandler`. In order for the protocol workflows to emit their protocol messages and take advantage of this design, they are required to have a reference to the server object; so they can easily call the correct emit functions. Alternatively, the workflows need to have access to and make a call to a given callback reference that calls the desired emit function in the server. The second alternative here is quite useful when the operations

7.2 Workflow Details

for the protocol workflow are initialized in the server, making it easy to add a callback reference as an initializer parameter. The reactive operations for the checkpoints and view-change can potentially be initialized in the server, making it simple to assign the correct callback function. An example of this is seen in Listing 7.9 and Listing 7.5 and are discussed in more detail later in Section 7.2

Due to our goal of testing the Cleipnir reactive framework, we have deliberately chosen to use the reactive framework every time our protocol workflows needed to wait for and handle protocol messages. This means the functionality for listening in for desired protocol messages and the functionality used to make valid certificates are handled using Cleipnir reactive framework. In addition, in certain areas, we have used Cleipnir reactive framework to implement event-handlers that are set to activate certain processes once a signal or item is sent to the desired process `Source` object.

Our PBFT implementation takes advantage of traditional asynchronous programming for the network layer. We chose this design primarily due to asynchronous programming being generally preferred for multi-client server design [8], [12]. Considering a replica needed to handle multiple client requests and protocol messages from the other replicas, this seemed like the best choice. The network layer does not take advantage of Cleipnir reactive programming and persistency functionality. Therefore we do not have to worry about `CTask` either, meaning the network functionality all uses traditional `Task`.

7.2 Workflow Details

7.2.1 Protocol Workflow Implementation

7.2.1.1 Starting protocol instance

A normal sequence for the PBFT implementation begins once the request handler receives a request message from the server. The source code for the request handler can be seen in Listing 7.1. The request handler listens for new requests messages emitted to the `Source` object `requestMessage` as seen on line 3. As mentioned in Section 6.4, the server is tasked with emitting messages received in the network layer to the appropriate `Source` object for the protocol to access the message. The request handler is responsible for making sure that the request received is valid. In addition, the request handler only starts a new iteration of the PBFT protocol when the next sequence number is within the current sequence number interval. This condition is handled by the if condition spanning lines 4-10. Finally, a new protocol instance is not initialized when the system is performing a view-change. This is determined by the boolean value `active` which is tied to the protocol execution object. Once all checks are passed, the request handler updates and collects the current sequence number. Then it calls the asynchronous `CTask` function `PerformProtocol` which initializes and starts the PBFT protocol for the given request. It is important that the request handler is not forced to wait for the `PerformProtocol` function to finish because the application must have access to the `requestMessage Source<Request>` object.

7.2 Workflow Details

This is because we desire an application that can process multiple requests from clients at the same time. If the application does not have access to the *requestMessage* for a long period, then it is likely that a request message emitted by the server gets lost.

```
1 while (true)
2 {
3     var req = await requestMessage.Next();
4     if (Crypto.VerifySignature(
5         req.Signature,
6         req.CreateCopyTemplate().SerializeToBuffer(),
7         serv.ClientPubKeyRegister[req.ClientID]
8     )
9     && serv.CurSeqNr < serv.CurSeqRange.End.Value
10    )
11    {
12        if (execute.Active)
13        {
14            int seq = ++serv.CurSeqNr;
15            Console.WriteLine("Curseq: " + seq + " for request: " + req);
16            _ = PerformProtocol(execute, serv, scheduler, shutdownPhaseSource, req,
17                               seq);
18        }
19    }
```

Listing 7.1: Code section for the request handler

7.2.1.2 Pre-Prepare phase

The pre-prepare phase is the only part of the normal operation workflow that has a different structure depending on whether or not the replica is the primary replica. The source code for the primary replica's pre-prepare phase can be seen in Listing 7.2. If the replica is the primary, it uses the sequence number that the protocol instance was initialized with and creates a pre-prepare message for this sequence number. The pre-prepare message also contains information regarding the primary's server id, current view, and the request digest. The pre-prepare message dictates the other replicas sequence number for the processing of the given request. The primary then initializes the protocol certificate used for storing the proof of the prepare phase. Since the first received phase message in the prepare phase is always supposed to be the pre-prepare message, the protocol certificate used for the prepare phase always has the pre-prepare message as its first entry in its proof list. The protocol instance then uses the server reference to multicast the pre-prepare message to the other replicas in the network.

The source code for the pre-prepare phase for the non-primary replicas is shown in Listing 7.3. The non-primary replica starts its protocol instance by subscribing to the `Source<PhaseMessage> MesBridge` and listens for incoming phase messages. The subscribe, listening, and handling process of the incoming items to the *MesBridge* is performed on lines 3-12. Considering the replicas only want a pre-prepare message in this reactive listener, it uses a `WHERE` clause to ignore any other phase message other than ones that use the pre-prepare messages enum type. In addition, another `WHERE` clause is assigned to avoid any pre-prepare messages designated for other requests by comparing request digests. Therefore an incoming phase message can only pass the `WHERE` clause if it involves the same request which the protocol instance is processing. The final `WHERE`

7.2 Workflow Details

clause validates the phase message where the validation criteria are the same as the ones mentioned in Section 4.3 for pre-prepare messages. Once the replica receives a pre-prepare phase message which passes all the `WHERE` clauses, it creates a protocol certificate that uses the same sequence number as the primary's pre-prepare phase message. The protocol certificate for the prepare phase now has a matching sequence number for each replica. The non-primary replica finally ends the pre-prepare phase. The normal workflow implementation starts the prepare phase by creating a prepare message and multicasting this message using the same method the primary used for multicasting its pre-prepare phase.

The `MERGE` operator is used to ensure that the protocol execution is terminated if a view-change occurs. If the timeout occurs, a unique phase message is emitted to the `Source<PhaseMessage> ShutdownBridgePhase`. The `MERGE` operator binds `ShutdownBridgePhase` reactive stream together with the `MesBridge` stream. This means it is now possible for the `MesBridge` to be unsubscribed without having to pass the previous operators in the reactive chain. This scenario only applies whenever a phase message is detected in the `ShutdownBridgePhase`. As the `Merge` operator is the last reactive operator in the chain, the stream returns the phase message received from the `ShutdownBridgePhase` as the resulting phase message. This phase message is intentionally faulty and is not allowed to be used in the prepare phase of the protocol. Therefore once this faulty phase message is received, a timeout exception is instead called, which closes the instance of the protocol execution.

The design chosen for the source code to the pre-prepare phase is simple and follows a synchronous workflow as we desired, making it easier for developers to write. Unfortunately, there are two severe issues with our current implementation of the pre-prepare phase. These issues are caused by a combination of splitting the code based on primary versus non-primary and the importance of initializing instances of the reactive listeners early. Both problems are theoretically very similar as they both are caused by improper initialization of the reactive listeners used in the PBFT implementation. The first issue occurs when the primary sends out its pre-prepare phase message before the non-primary replicas have initialized the pre-prepare reactive listener. This results in the pre-prepare phase message not being received by the non-replica, which means it fails the pre-prepare phase. As the pre-prepare phase fails, the timeout will eventually occur, which puts the replica into view-change mode as it believes that the primary replica is faulty. The second issue is that a non-primary can receive a prepare message before it has received the initial pre-prepare message from the primary. When this situation occurs, the prepare message gets filtered out by the pre-prepare reactive listener and is therefore not available once this non-primary reaches the prepare phase. In the worst-case scenario, the replica loses all of the prepare phase messages from the other replicas, meaning the protocol instance is stuck in the prepare phase once it finally receives its pre-prepare message.

These issues just discussed are caused primarily because the application struggles with handling phase messages that are received out of intended order. There exist several workarounds to handle messages that arrive out of order. However, most of the workarounds available would require adding a lot more complexity to the implementation. As our goal for this thesis is to create an PBFT implementation that is very simple and accurate to the protocol description, we decided not to redesign the protocol workflow to handle issues regarding pre-prepare messages out of order. As it is meant for the pre-

7.2 Workflow Details

prepare message to get the other non-primary to start processing the request by providing the correct sequence number, we feel it would not be faithful to the original algorithm to change this design. Once the pre-prepare message is received, the reactive listener for the prepare messages that did not have the same sequence number that matches the received pre-prepare message would be filtered out. Currently, to somewhat mitigate this issue, the primary is forced to wait for at least a second before starting to multicast its pre-prepare message. Performing this waiting period allows the other replicas to catch up. Which makes it less likely that a replica is far enough behind to lose out on prepare messages before completing handling their pre-prepare message. With this workaround, the issues discussed here are, for the most part, stable. As an estimate, an average of 15 operations can be progressed without incident before a user encounters these issues.

```
1 ProtocolCertificate qcrtpre;
2 byte[] digest = Crypto.CreateDigest(clireq);
3 int curSeq;
4 if (Serv.IsPrimary()) //Primary
5 {
6     curSeq = leaderseq;
7     Console.WriteLine("CurSeq:" + curSeq);
8     Serv.InitializeLog(curSeq);
9     PhaseMessage preprepare = new PhaseMessage(
10         Serv.ServID,
11         curSeq,
12         Serv.CurView,
13         digest,
14         PMessageType.PrePrepare
15     );
16     Serv.SignMessage(preprepare, MessageType.PhaseMessage);
17     qcrtpre = new ProtocolCertificate(
18         preprepare.SeqNr,
19         preprepare.ViewNr,
20         digest,
21         CertType.Prepared,
22         preprepare
23     );
24     await Sleep.Until(1000);
25     Serv.Multicast(preprepare.SerializeToBuffer(), MessageType.PhaseMessage);
26 }
```

Listing 7.2: Source code for pre-prepare phase for primary replica

7.2 Workflow Details

```
1  else    //Not Primary
2  {
3      var preprepared = await MesBridge
4          .Where(pm => pm.PhaseType == PMessageType.PrePrepare)
5          .Where(pm => pm.Digest != null && pm.Digest.SequenceEqual(digest))
6          .Where(pm => pm.Validate(
7              Serv.ServPubKeyRegister[pm.ServID],
8              Serv.CurView,
9              Serv.CurSeqRange)
10         )
11         .Merge(ShutdownBridgePhase)
12         .Next();
13
14     if (preprepared.ServID == -1 && preprepared.PhaseType == PMessageType.End)
15         throw new TimeoutException("Timeout Occurred! System is no longer active!");
16     qcertypre = new ProtocolCertificate(
17         preprepared.SeqNr,
18         Serv.CurView,
19         digest,
20         CertType.Prepared,
21         preprepared
22     );
23     curSeq = qcertypre.SeqNr;
24     Serv.InitializeLog(curSeq);
25     PhaseMessage prepare = new PhaseMessage(
26         Serv.ServID,
27         curSeq,
28         Serv.CurView,
29         digest,
30         PMessageType.Prepare
31     );
32     Serv.SignMessage(prepare, MessageType.PhaseMessage);
33     qcertypre.ProofList.Add(prepare);
34     Serv.Multicast(prepare.SerializeToBuffer(), MessageType.PhaseMessage);
35 }
```

Listing 7.3: Source code for Pre-prepare phase for non-primary replica

7.2.1.3 Prepare phase

In comparison to the Pre-prepare phase and the start of the prepare phase, the rest of the workflow in the implementation is relatively stable and straightforward. The prepare and commit phase source code can be seen in Listing 7.4. The first step of the prepare phase is to initialize the reactive listeners for prepare and commit phase messages. Due to the listeners having several reactive operators connected to their stream, the code must span several code lines to make it more readable. The prepare listener is initialized on lines 2-18, and the commit listener is initialized on lines 25-42 in Listing 7.4. There are two reasons why the reactive listeners for prepare and commit messages are initialized early. The first reason is to reduce the time it takes for the workflow to move from the pre-prepare listener to the following reactive listeners. This time needs to be small to avoid losing potential incoming phase messages to the reactive streams. The other reason is to avoid ordering issues between prepare and commit messages. Since the sequence number for the workflow has already been determined during the pre-prepare phase, the prepare and commit phase can initialize their reactive streams early and be active simultaneously. Because of this, the prepare and commit phase does not suffer issues in regards to phase messages being out of order. If the pre-prepare message did not dictate the sequence number for non-primary replicas, this would have also been the ideal design for handling phase messages during the pre-prepare phase.

7.2 Workflow Details

The reactive listeners used for the prepare phase and the commit phase are almost practically identical. The only significant difference between the two reactive listeners is that they only accept phase messages in the stream with their respective protocol phase. For example, the reactive listener for the prepare phase filters away phase messages that do not have protocol phase-type prepare. This operation is performed by the first **WHERE** clause. In addition, the certificates for both protocol phases are also initialized early. This is because the certificates are now actively updated through the operations in the reactive listeners' stream instead of returning the emitted phase message.

During the prepare phase, the workflow waits until the prepare certificate has added $2f + 1$ unique prepare phase messages to its proof list. For a phase message to be added to the prepare certificate, it must pass all of the **WHERE** clauses assigned for the reactive listener. In actuality, the workflow only waits for $2f$ prepare phase messages due to the pre-prepare message already been added to the protocol certificate during the pre-prepare phase. Once a valid phase message passes all of the first **WHERE** operators, it is added to the designated protocol certificate using the **SCAN** operator. The **SCAN** operator transforms the certificate's proof list to include the incoming phase message. The final **WHERE** clause determines whether or not the certificate has reached a sufficient number of valid phase messages in its proof list. The `ValidateCertificate` function essentially calculates the number of phase messages inside the proof list when it excludes duplicates. It also makes sure that the phase messages in the list are indeed valid. The asynchronous `await` operator on line 45 is used to wait for the `CAwaitable` in the prepare phase reactive listener to finish all of the linked operators for the listener before moving on with the protocol. Once the validation process has succeeded for the protocol certificate, the workflow can move past the `await` operator. The prepare phase finishes after the prepare protocol certificate is added to the protocol log in the server on line 47.

7.2.1.4 Commit Phase

As for the commit phase, like the other protocol phases, the first step is to have each replica create a commit phase message and use the server to multicast the commit phase over the PBFT network. Afterwards, the commit phase performs practically the same operations as the prepare reactive listener. The commit reactive listener waits for the proof list for the commit certificate to have at least $2f + 1$ commit phase messages. The reactive listener for the commit phase has an additional **WHERE** clause that makes sure that the prepare phase has already finished before exiting the commit reactive listener, which is visible on line 41 in Listing 7.4. This extra **WHERE** clause is used to avoid the commit certificate from being finished before the prepare phase is complete. After the commit certificate is successfully validated, the protocol workflow is almost finished processing the given request. The protocol workflow first adds the commit certificate to the protocol logger as done prior to the prepare certificate before starting the remaining operations in the protocol workflow. The server now has two valid certificates for the given sequence number assigned to the client request, meaning the replica has the necessary proof that the replicas in the PBFT network agree to have the application perform the operation for the given request. The application finally performs the operation within the request. The last remaining process is to create a reply message, digitally sign this reply message

7.2 Workflow Details

and send the reply message to the client who initially sent the processed request. The reply message includes information to the client in regards to whether the operation given in the original request was completed successfully or not. In our PBFT implementation, the only operation the application can do is to write the ‘operation’ received from the request to the console window and add the operation to a persistent list. The persistent list representing the application state is discussed more in Section 7.2.2.

7.2.1.5 Protocol Workflow Evaluation

We succeeded in our objective of creating an implementation that performs the standard processes of the PBFT algorithm into a single function. We believe our resulting implementation is relatively faithful to the PBFT protocol when based only on the protocol description. The majority of the operations performed in the normal protocol workflow are synchronous, making it easier to read the code. The reactive `Source` objects and their chain of operators are the only operations that are performed asynchronously in the normal protocol workflow. The reactive operators still work well together with the synchronous workflow by completing the required checks and operations on the incoming phase messages independently from the rest of the protocol workflow. By taking advantage of the `await` command, we easily mark the areas in the protocol workflow where we know the protocol workflow cannot function without the result from the reactive operators. We believe the most significant benefit for our design in the PBFT protocol workflow within a single function is that it became a lot easier to keep track of the protocol operations. For example, due to how the implementation handles the workflow that creates the protocol certificates, it is considerably straightforward to differentiate between the PBFT protocol phases by looking at the source code. Basically, by looking for the `await` points in the protocol workflow, we can approximately determine where one of the three protocol phases finishes. It is an approximation since there are still a couple of operations required to be performed, such as adding the certificate to the protocol log for the prepare and commit phases. We would argue that it is a significant challenge to simplify the code to prepare and commit phases further without causing a severe issue for protocol workflow. Most of the complexity we currently have in our implementation comes from the fact that the primary and non-primary replicas have different operations in the pre-prepare phase. In addition, our current stop functionality is not exactly straightforward, which further hurts the simplicity of our implementation. We also believe that despite the functionality of the reactive operators being convenient for handling protocol messages, they may be difficult for inexperienced programmers to read. The programmers should at the very least have some fundamental knowledge in regards to chaining operators using query languages such as SQL, preferably knowing the fundamentals of LINQ [50] statements, to fully grasp most of the reactive operations available in the Cleipnir framework. The normal workflow implementation is around 135 lines of code when we exclude any additional spaces used to make object initialization easier for others to read. In addition, approximately 30% of the lines are used for the reactive operators. All in all, based on these results, we would argue that the implementation is relatively short to be able to handle all three protocol phases inside a single function. However, an apparent problem with keeping the functionality in this format was the difficulty of handling protocol messages out of order, which forces the developer to deploy workarounds to avoid this

7.2 Workflow Details

problem. We choose to initialize the `Source` objects as soon as possible to reduce the number of phase messages dropped. Unfortunately, we cannot deal with the pre-prepare phase messages due to only being used by non-primary replicas, which is a big downside to using the desired format with reactive operators.

Due to us performing the protocol workflow inside `CTASK` functions, we are not able to use traditional asynchronous operations inside the protocol workflow. This typically means tasks regarding reading data from files, server requests, networking, or any other job that is preferred to be performed asynchronously should do so outside the protocol workflow. A developer would therefore need to keep this in mind when designing the protocol-related workflows. Still, regardless of whether or not the protocol workflow is performed inside a `Task` or a `CTask`, the workflow is run asynchronously, allowing us to effortlessly run instances of the protocol workflow separately. Although, like when using threads, we need to ensure that the separate asynchronous functions do not alter the same properties if the execution order matters. Otherwise, the result of the application state would become unpredictable. Although our PBFT implementation does add the two resulting protocol certificates to a shared log, the sequence number assigned to the protocol workflow is unique for each iteration of the PBFT workflow, allowing us to avoid corrupting the protocol state. This is because all of the iterations will have a unique key to store their protocol information. Despite being run in a `CTask`, the `await` operator is still valuable for the protocol workflow as it is used to wait for the certificates to finish. Without the ability to use the `await` operator, we would not have been able to create the desired protocol workflow. All in all, the asynchronous workflow may hinder the developer from performing certain operations directly inside the workflow. It is still beneficial when looking at the benefits of running the protocol workflow asynchronously. The most significant advantage of running the workflow asynchronously is how simple it is to start multiple iterations of the protocol workflow. In addition, running the protocol workflow asynchronously should scale better for various clients in comparison to creating separate threads.

Cleipnir reactive framework was handy for handling protocol workflow received from the server. Although we did have quite the big issue with the usage of `Source` objects to create the protocol certificate before using the Cleipnir execution engine to schedule the emits. Once we moved on to using universal formatting for emitting items to the workflows in Cleipnir, it has worked as intended. It was initially challenging to use a couple of Cleipnir reactive operators such as `Merge` and `Scan`, but relatively simple to learn the general approach of chaining reactive operations. Using `Source` objects for sending the protocol messages to their appropriate code section is simple once the Cleipnir execution engine was used to schedule the emits in order. From our experience keeping the emit functionality centred to a designated object or class that has reference to the execution engine is the recommended structure. Although, we must consider one aspect when we use reactive programming to handle protocol messages. Each protocol message, regardless of the owner, must all be sent to the protocol workflow in the same way. In our case, all phase messages had to be emitted to the `Source` object for the phase message to be validated correctly. This includes its own phase message, meaning a functionality must be available for the protocol workflow to emit phase messages created during protocol workflow. The main advantage of using Cleipnir reactive framework to handle protocol messages for the protocol workflow is that we can structure all of the

7.2 Workflow Details

code related to the phase message inside a single block of code. In addition, due to the nature of chaining operators together in a reactive chain, it is easy to control the order of the operations that need to be on the protocol message. Finally, combined with the `await` operator, we can also easily dictate wherein the workflow we want to wait for the sufficient number of protocol messages to be received until the condition required by the consensus algorithm is met. In our case, this would substitute for the process of creating valid protocol certificates by submitting validated proofs until reaching the quota. To summarize the main benefits, the Cleipnir reactive framework provided our normal PBFT workflow implementation was a simple way to validate and collect phase messages to create valid protocol certificates. In addition, to making the collaboration between the network layer and the protocol layer easier to develop. The downsides being that they struggle with handling phase messages that are received out of order. Therefore, as a consequence must be initialized as soon as possible to counteract this issue.

7.2 Workflow Details

```
1 var prepared = MesBridge
2     .Where(pm => pm.PhaseType == PMessageType.Prepare)
3     .Where(pm => pm.SeqNr == qcrtpre.SeqNr)
4     .Where(pm => pm.Validate(
5         Serv.ServPubKeyRegister[pm.ServID],
6         Serv.CurView,
7         Serv.CurSeqRange,
8         qcrtpre)
9     )
10    .Where(pm => pm.Digest.SequenceEqual(qcrtpre.CurReqDigest))
11    .Scan(qcrtpre.ProofList, (prooflist, message) =>
12        {
13            prooflist.Add(message);
14            return prooflist;
15        })
16    .Where(_ => qcrtpre.ValidateCertificate(FailureNr))
17    .Next();
18 ProtocolCertificate qcrtcom = new ProtocolCertificate(
19     qcrtpre.SeqNr,
20     Serv.CurView,
21     digest,
22     CertType.Committed
23 );
24 var committed = MesBridge
25     .Where(pm => pm.PhaseType == PMessageType.Commit)
26     .Where(pm => pm.SeqNr == qcrtcom.SeqNr)
27     .Where(pm => pm.Validate(
28         Serv.ServPubKeyRegister[pm.ServID],
29         Serv.CurView,
30         Serv.CurSeqRange,
31         qcrtcom)
32     )
33     .Where(pm => pm.Digest.SequenceEqual(qcrtcom.CurReqDigest))
34     .Scan(qcrtcom.ProofList, (prooflist, message) =>
35        {
36            prooflist.Add(message);
37            return prooflist;
38        })
39     .Where(_ => qcrtcom.ValidateCertificate(FailureNr))
40     .Where(_ => qcrtpre.ValidateCertificate(FailureNr))
41     .Next();
42
43 Console.WriteLine("Waiting for prepares");
44 if (Active) await prepared;
45 else throw new ConstraintException("System is no longer active!");
46 Serv.AddProtocolCertificate(qcrtpre.SeqNr, qcrtpre); //add first certificate to Log
47
48 //Commit phase
49 PhaseMessage commitmes = new PhaseMessage(
50     Serv.ServID,
51     curSeq,
52     Serv.CurView,
53     digest,
54     PMessageType.Commit
55 );
56 Serv.SignMessage(commitmes, MessageType.PhaseMessage);
57 Serv.Multicast(commitmes.SerializeToBuffer(), MessageType.PhaseMessage);
58 Serv.EmitPhaseMessageLocally(commitmes);
59 Console.WriteLine("Waiting for commits");
60 if (Active) await committed;
61 else throw new ConstraintException("System is no longer active!");
62 Serv.AddProtocolCertificate(qcrtcom.SeqNr, qcrtcom); //add second certificate to Log
```

Listing 7.4: Source code for Prepare and Commit phase

7.2.2 Checkpoint Implementation

The checkpointing process only occurs after a certain number of requests have been processed by the PBFT implementation. The *checkpoint interval* determines the number of requests. For our implementation, the *checkpoint interval* is set to five, meaning after processing five requests, a new checkpoint is created for the system. Our implementation of the checkpoint workflow is divided into three sections. The first section revolves around creating a checkpoint certificate and starting an instance of the reactive checkpoint workflow. The reactive checkpoint workflow performs the second part of the checkpoint workflow. In this part, a reactive `Source` object listens for incoming checkpoint messages, which are then validated and added to the checkpoint certificate's proof list. This reactive process ends once a certificate has received sufficient checkpoint messages that are deemed valid. The final part consists of emitting the finished stable checkpoint to the server to replace the last stable checkpoint in memory and start the garbage collection process. We are now going to discuss each of these parts in more detail.

7.2.2.1 Initialize Checkpoint Certificate

The checkpoint certificate is initialized using the last sequence number used by the protocol workflow. The checkpoint certificate also needs to create and store a digest of the current state of the application. Our implementation makes the system digest based on the persistent list that represents the application state. The persistent list contains the operation messages from each of the fully processed requests by the PBFT protocol. Therefore assuming no errors occur, then the checkpoint for sequence number five has the digest of the list containing the operation from requests one to five.

The checkpoint workflow starts by first initializing the checkpoint certificate. The certificate includes the information just described, such as the stable sequence number and the digest of the application state. Once the initialization of the checkpoint certificate is done, the checkpoint workflow starts an instance of the checkpoint reactive workflow for the newly created checkpoint certificate. We refer to an instance of checkpoint reactive workflow process as a *Checkpoint Listener*. Additionally, the checkpoint certificate is added to the checkpoint logger using the stable sequence number as the key. The process just described can be started in two separate ways. The first method is when the replica itself actively starts the checkpoint process. This is when the replica has processed enough requests in the PBFT workflow to reach the checkpoint interval. The other approach is when the replica receives a checkpoint message with a sequence number currently not in the checkpoint logger. The checkpoint logger also needs to verify that the checkpoint message has a higher sequence number than the last stable checkpoint stored on the replica. Both methods perform the initialization of the checkpoint certificate and checkpoint listener. However, the sequence number used for the initialization process differ. The first method uses the last sequence number the protocol processed that initially triggered the checkpoint process. The other way uses the sequence number from the received checkpoint message. One thing to remember is that the replica only performs this process only once for a sequence number. Meaning the protocol logger is checked to determine whether or not the checkpoint certificate has already been initial-

7.2 Workflow Details

ized or not. If the checkpoint certificate is already stored in the logger, the initialization process is not performed again.

Regardless of the way the checkpoint certificate and listener are initialized, the replica is still required to create and multicast its checkpoint message to the PBFT network once the sequence number matches a checkpoint interval. The checkpoint message created in the replica is also emitted to the checkpoint listener to allow it to be handled the same way as the other checkpoint messages received from the PBFT network. The checkpoint certificates initially stored in the checkpoint logger are not stable checkpoint certificates, and our goal is to make at least one of these certificates stable. However, for a checkpoint to be deemed stable, it needs to pass the certificate validation processes in the checkpoint listener, which follow the same guidelines as the protocol certificate. A replica can only store one stable checkpoint, meaning the previous stable checkpoint is overwritten whenever a new stable checkpoint with a higher sequence is available. The stable checkpoint certificate is used as definitive proof that the PBFT network agrees on the state of the application up to the stable sequence, meaning the replicas in the PBFT network now can garbage collect the protocol data from the logger up to the stable sequence number. The garbage collection includes removing any stored checkpoint certificates in the checkpoint logger with lower or equal sequence numbers to the stable checkpoint certificate.

7.2.2.2 Checkpoint Listener Workflow

The source code for an instance of a checkpoint listener is presented in Listing 7.5. The checkpoint listener uses `Source<Checkpoint>` similar to how `Source<PhaseMessage>` objects were used in the protocol workflow. The server, once it receives a checkpoint message from the network, emits the checkpoint message to the `Source<Checkpoint>` shared by the server and the checkpoint listeners. The checkpoint listener listens for checkpoint messages emitted by the server to the `Source<Checkpoint>` object. The reactive operations performed on the `Source<Checkpoint>` object can be seen on lines 8-17. The checkpoint message received on the stream is first validated before the checkpoint certificate proof list is transformed to have the checkpoint message in its proof list. The `WHERE` clauses on line 9 and 10 performs the validation for incoming checkpoint messages. The `SCAN` operator is once again used to add the checkpoint to the certificate proof list.

Unlike the protocol workflow, the iterations of the checkpoint listeners do not need to finish their execution. In addition, the checkpoint functionality is performed separately from the protocol workflow, meaning the protocol workflow can process new requests while the checkpoint workflow tries to create a stable checkpoint certificate. Assuming the protocol workflow has not exceeded the sequence number interval otherwise, no additional requests are processed by the protocol workflow. If the protocol workflow processes enough requests, a new checkpoint listener is created for another checkpoint with a higher sequence number than the preceding one. This means it is possible to have multiple checkpoint listeners active at the same time. However, it then becomes a race for the checkpoint listeners to see which one creates the next stable checkpoint certificate. Although it is important to remember that the system does not process any new requests after it has

7.2 Workflow Details

exceeded the current sequence number interval.

The reactive listener is finished when all of the reactive operators for the `Source<Checkpoint>` have ended, which requires the checkpoint certificate to be stable. A checkpoint certificate is deemed stable once it has $2f + 1$ unique and valid checkpoint messages in its proof list. The checkpoint messages in the proof list must match the checkpoint certificate sequence number and digest, which are checked during the certificate validation in the `WHERE` clause on line 16.

```
1 public async CTask Listen(  
2 CheckpointCertificate cpc,  
3 Dictionary<int, RSAParameters> keys,  
4 Action<CheckpointCertificate> finCallback  
5 )  
6 {  
7     Console.WriteLine("Checkpoint Listener: " + StableSeqNr);  
8     await CheckpointBridge  
9     .Where(check => check.StableSeqNr == StableSeqNr)  
10    .Where(check => check.Validate(keys[check.ServID]))  
11    .Scan(cpc.ProofList, (prooflist, message) =>  
12        {  
13            prooflist.Add(message);  
14            return prooflist;  
15        })  
16    .Where(_ => cpc.ValidateCertificate(FailureNr))  
17    .Next();  
18    finCallback(cpc);  
19 }
```

Listing 7.5: Source code for the Checkpoint Listener

7.2.2.3 Initiate Garbage Collection

The third part of the checkpoint functionality is rather simplistic. During startup, the replica initializes its server functionality, including an asynchronous function that listens on a reactive `Source<CheckpointCertificate>`. This reactive listener listens for a new stable checkpoint certificate. Once the `Source<CheckpointCertificate>` object receives a stable checkpoint certificate, the current stable checkpoint is overwritten by the one it received. Afterwards, the operations in regards to garbage collection are performed. The source code for listening for stable checkpoint certificate can be seen in Listing 7.6 The `Source<CheckpointCertificate>` object connected to this function is persisted on the server. The server has a predefined function that uses the Cleipnir scheduler to schedule an emit to this `Source` object. Each checkpoint listener is initialized with the callback reference to this function, which allows the checkpoint listener to immediately call the callback address with the finished stable checkpoint certificate whenever all reactive operations are done. The call on the callback reference can be seen in Listing 7.5 on line 18. The Cleipnir execution engine then schedules the stable checkpoint to be emitted to the reactive listener for stable checkpoint certificates. Once the `Source` object receives, the old stable checkpoint certificate is replaced by the new one, even in the case where the replica does not have any existing stable checkpoint certificates. After the new stable checkpoint certificate is assigned to the replica, the garbage collection process begins. The garbage collection process consists of removing records with a lower or equal sequence number to the new stable checkpoint certificate for the protocol, reply,

7.2 Workflow Details

and checkpoint logger. After the garbage collection is completed, the sequence number interval is extended, allowing the protocol workflow to process more requests.

```
1 public async CTask ListenForStableCheckpoint ()
2 {
3     Console.WriteLine("Listen for stable checkpoints");
4     while (true)
5     {
6         var stablecheck = await Subjects.CheckpointFinSubject.Next ();
7         Console.WriteLine("Update Checkpoint State");
8         Console.WriteLine(stablecheck);
9         StableCheckpointsCertificate = stablecheck;
10        GarbageCollectLog (StableCheckpointsCertificate.LastSeqNr);
11        GarbageCollectReplyLog (StableCheckpointsCertificate.LastSeqNr);
12        GarbageCollectCheckpointLog (StableCheckpointsCertificate.LastSeqNr);
13        UpdateRange (stablecheck.LastSeqNr);
14    }
15 }
```

Listing 7.6: Reactive handler for new stable checkpoints

7.2.2.4 Checkpoint Workflow Evaluation

Unfortunately, unlike the normal protocol workflow, we could not keep the checkpoint workflow centred around a single function or class. The main challenge design-wise for the checkpoint workflow was the fact that the checkpoint process could be initialized by any replica in the PBFT network. The checkpoint workflow needed to handle both initialization methods in addition to having the same workflow regardless of whether or not the process was initialized by a received checkpoint message or by the checkpoint interval. Ultimately since the server network layer received the checkpoint messages, we decided it was best to divide up the workflow and instead initialize the certificate and listener process only if no record existed for the stable sequence number. Regardless we would argue that design-wise, we managed to divide up the program in such a way that the process itself remains simple. It is not more complicated than summarizing the checkpoint workflow as initialization, listening, and initiate garbage collection. Of course, it is a lot more difficult when looking at individual operations in more detail.

Due to the checkpoint processes being performed wholly separate from the rest of the protocol workflow, running most of the checkpoint workflow asynchronously was important. The checkpoint initialization process is only part of the checkpoint workflow that was performed synchronously. Both the checkpoint listener and the listen for stable checkpoint certificate take advantage of both asynchronous programming and reactive programming. Both parts of the workflow are required to wait until the desired criteria are met. The checkpoint listener is ideal for asynchronous workflow because it is performed independently from the protocol workflows. In addition, it is unclear when or if an instance of the checkpoint listener would ever finish. Therefore, it was crucial to make sure the checkpoint listener does not block the thread or steal unnecessary resources. Regarding the last part of the checkpoint functionality, it is required to be practically active all the time as it is unclear when a new stable checkpoint is ready. However, since the garbage collection process rarely occurs and most of the time, the checkpoint functionality is simply waiting, thereby making it desirable to use asynchronous workflow here.

Initially, we did not take much advantage of reactive programming when we designed the process of making a checkpoint certificate stable by adding proofs to it. However, changes were made to the design to accommodate for more reactive programming. The result being the checkpoint listener workflow we shown in Listing 7.5, which we described in Section 7.2.2.2. The original implementation performed all checkpoint message validations and certificate checks whenever a checkpoint message was being added to the proof list. We managed this functionality by using a designated append function, which performed the same operations that are now performed in the reactive chain in the checkpoint listener. The original implementation was functional; however, it was also relatively unstable, meaning we had many additional conditions to check based on where the append function was called. The primary benefactor to the issues came with the usage of the callback functionality. Not only did we have to assign a callback reference as part of the checkpoint certificate, but the call process also became somewhat unpredictable. A significant contributor was that it was common to call the append function more times than necessary due to receiving more checkpoint messages than was needed. Combine this with the short time intervals between each checkpoint message, and you will get unpredictable results. Not to mention, the Cleipnir execution engine had to schedule the append function calls to avoid the state of the checkpoint certificate becoming unpredictable. Safe to say, we generally preferred the second implementation, which is why it is the presented workflow. Generally, the design for the second implementation was a lot more readable and easier to keep track of the workflow. The second implementation also was a lot more stable due to splitting up the processes in the append function into separate reactive operators with more focus on completing a single task. It had better performance due to only having to use the Cleipnir execution engine to schedule the emit to the checkpoint listener rather than having to schedule all of the operations for each checkpoint message as we were forced to with our original design. Finally, the checkpoint certificate no longer needed to have a record of the callback address to the desired emit function in the server. Instead, it was added as a parameter when the checkpoint listener started listening. The garbage collector functionality has remained the same for both implementations and uses Cleipnir reactive `Source` object similar to how channels are used in Golang programming. Generally, this structure works well for the garbage collector because emits only occurs whenever a new stable checkpoint certificate is ready.

7.2.3 View-change Implementation

As previously mentioned in Section 4.5 the goal of a view-change is to replace a faulty primary replica with another non-faulty replica successfully. For a view-change to be successful, the replicas in the PBFT network must agree upon a protocol state that each replica can move on from after the leader change has occurred. Furthermore, the view-change must ensure that the newly selected primary replica is not also faulty. The implementation of the view-change functionality is a lot more complex in comparison to the normal protocol workflow. Several aspects make the view-change functionality challenging to handle appropriately. The view-change must first have some functionality to stop the normal protocol workflow, even when the protocol is still processing a request. Afterwards the view-change messages are exchanged over the PBFT network until $2f + 1$ replicas agree that the system needs to change view. Finally, the replicas have to

reprocess any protocol certificates saved in the protocol logger. Our implementation of the view-change can be better described by dividing the workflow into three segments. The first part consists of starting the view-change process. This includes the functionality for stopping active protocol instances. In this section, the application is also set to ignore future protocol messages received during the view-change process. The second part consists of updating the replica's view information and creating and multicasting a view-change message to the PBFT network. The second part is also responsible for creating the view-change certificate. The last segment is the functionality in regards to setting up the correct protocol state of the PBFT network for the new view. In the following sections, we will describe the different parts of our view-change implementation in the order in which they are performed.

7.2.3.1 Starting a View-Change

A View-change is started whenever a replica deems the current primary to be faulty. In our implementation, a replica can determine that a primary is defective in two separate ways. The first is the more common approach. We use a timeout functionality to detect irregular activity for the primary replica. The other condition that can start a view-change for the replica is when the replica has received a total of $2f$ view-change messages from the other replicas in the PBFT network. In this situation, the replica knows that the view-change exchange only needs its own view-change message for it to be successful.

In our case, we only support timeout functionality in the protocol workflow during the period where a replica is waiting for a pre-prepare phase message from the primary for a request the replica previously has received. Listing 7.7 shows the source code for where the timeout functionality is initialized. Listing 7.7 also shows how we initialize and how we stop the overall protocol workflow. On line 9-12 we can initialize the `AppOperation` within a `WhenAny` asynchronous function. The `WhenAny` creates a `CTask` for the two asynchronous `CTask` operations `AppOperation` and `ListenForShutdown`. The `CTask` created for `WhenAny` finishes whenever either of the `CTask` has finished its operation. In our case the `ListenForShutdown` is simply waiting for the given `Source` object `ShutdownSubject` to receive an item which constitutes as a shutdown signal. When a timeout occurs for the protocol workflow, the timeout emits an item to the `ShutdownSubject`, which in turn results in `ListenForShutdown` finishes first. Each iteration of the protocol workflow is given a `CancellationToken` to stop the timeout functionality after it has received a pre-prepare message from the primary. The `CTask<bool>` result provided by the `WhenAny` is used to tell the workflow whether or not the `AppOperation` managed to finish or if the timeout occurred first. If the `AppOperation` finishes first, then the return value is true. Otherwise, a timeout has occurred, and the boolean value is false. If the boolean has false value, we set the application to be in what we refer to as inactive mode. In inactive mode, all requests and protocol-related messages such as phase messages and checkpoints are ignored. The application remains in inactive mode until all of the segments of view-change have been successfully completed.

After the application is set to be inactive, the application must also stop any active normal protocol workflows. Theoretically, it is possible to keep existing protocol iterations alive

7.2 Workflow Details

during and after a view-change occurs. However, it would be rather wasteful because the `CTasks` are never finished. The `CTasks` are never stopped because the reactive streams never finish all of their reactive operators. This, in turn, would unnecessarily drain the system of resources due to each time an item is emitted to the protocol `Source` objects, the old iterations would receive these items as well. The old protocol workflows would drop them quite quickly because the view number of the received message never matches the old protocols, which is one of the first `Where` clauses used for the reactive stream. All in all, keeping the old protocol workflows running is possible but would be unnecessary and would waste resources. For this reason, we thought it is preferable to terminate any active protocol process whenever a view-change occurred.

We accomplish this by having the application emit a clearly faulty phase message to a `Source<Phasemessage>` called `shutdownPhaseSource`. This `Source` object corresponds to the `Source<Phasemessage>` used in the `Merge` operator shown in Listing 7.3. As we mentioned earlier in the Section 7.2.1.2, once the protocol workflow iterations receive the faulty pre-prepare message, it exits the function by throwing, as well as catching, a `TimeoutException`. In the case where the system has already received $2f$ view-change messages, the system emits the shutdown signal to the same `ShutdownSubject Source` object we use in the `ListenForShutdown` `CTask` function. Meaning the initialization for the view-change functionality does remain the same regardless of the method used to initiate it. The details in regards to handling view-change messages are described in detail in Paragraph 7.2.3.2.1. Line 29 in Listing 7.7 is where the view-change functionality begins, and the `await` operator is used to make sure the view-change is completed before the protocol can go back to being active.

7.2 Workflow Details

```
1 CancellationTokenSource cancel = new CancellationTokenSource();
2 _ = TimeoutOps.AbortableProtocolTimeoutOperation( //starts timeout
3   serv.Subjects.ShutdownSubject,
4   10000,
5   cancel.Token,
6   scheduler
7 );
8 execute.Serv.ChangeClientStatus(req.ClientID);
9 bool res = await WhenAny<bool>.Of(
10   AppOperation(req, execute, seq, cancel),
11   ListenForShutdown(serv.Subjects.ShutdownSubject)
12 );
13 Console.WriteLine("Result: " + res);
14 if (res)
15 {
16   Console.WriteLine($"APP OPERATION {seq} FINISHED");
17   ...
18 }
19 else
20 {
21   if (execute.Active)
22   {
23     Console.WriteLine("View-Change starting");
24     execute.Active = false;
25     serv.ProtocolActive = false;
26     await scheduler.Schedule(() =>
27       shutdownPhaseSource.Emit(new PhaseMessage(-1, -1, -1, null, PMessageType.End)
28     ));
29     await execute.HandlePrimaryChange2();
30     Console.WriteLine("View-Change completed");
31     serv.UpdateSeqNr();
32     if (serv.CurSeqNr % serv.CheckpointConstant == 0 && serv.CurSeqNr != 0
33         || serv.StableCheckpointsCertificate == null && serv.CurSeqNr >
34             serv.CheckpointConstant
35         || serv.StableCheckpointsCertificate != null &&
36             (serv.StableCheckpointsCertificate.LastSeqNr + serv.CheckpointConstant) <
37             serv.CurSeqNr)
38       serv.CreateCheckpoint2(execute.Serv.CurSeqNr, PseudoApp);
39     execute.Active = true;
40     serv.ProtocolActive = true;
41     serv.GarbageViewChangeRegistry(serv.CurView);
42     serv.ResetClientStatus();
```

Listing 7.7: Handling timeout for the normal protocol workflow and initiate the View-Change process

7.2.3.2 View-Change functionality

Listing 7.8 shows the overall workflow for our implementation of the view-change functionality. The workflow shown in Listing 7.8 is responsible for initializing and keeping in order the two last segments of our view-change implementation. Meaning it is responsible for updating the view information for the replica. When the replica starts participating in the view-change process, it needs to both create and multicast the replica's view-change message to the PBFT network. In addition, the replica needs to start listening in for and handle any incoming view-change messages received from the PBFT network. Once a valid view-change certificate is made, the replica starts the new-view phase of the view-change workflow.

A view-change can pick a new non-faulty primary as its leader due to the next primary being solely dependent on the $p = v \bmod R$ formula. Therefore we needed to implement a functionality that could restart the view-change process indefinitely if the view-change

exchange or new-view process were to fail or take too long. To handle this functionality, we currently use a mix of timeout operations and `goto` statements to reroute the program flow back to the beginning of the view-change process [51]. Specifically, by restarting the view-change workflow whenever a timeout occurs, we force the view-change functionality to keep updating its view information. Therefore, the view number is incremented each time the view-change protocol restarts, meaning the new primary chosen for the new view is continuously being swapped until a non-faulty primary is finally chosen. The timeout operations are initialized and used the same as they were for stopping the normal protocol workflow. This includes initializing them with cancellation tokens so that they can be stopped once the workflow has succeeded in performing the desired operation. Both the view-change exchange and the new-view process have their respective timeout operation. The view-change exchange has a timeout set to 10 seconds, just like the normal workflow, while the new-view process has a timeout set to 15 seconds. Extra time is added for the new-view process since it needs to reprocess at worst-case five requests for our implementation. The worst-case scenario number is determined by the checkpoint interval, which is set to five requests for our case. Therefore, the protocol logger could only have up to four finished requests where the last request is never fully processed by the protocol workflow. The `WhenAny` asynchronous function is once again used together with the timeout operations. If the view-change process is successful, the program moves on as intended. In the case where the timeout occurs first, then the `goto` statement moves the program back to the `ViewChange` label we initialized at the very first line of the view-change workflow. Just like in the checkpoint workflow, we refer to the functionality that uses `Source` object to listen for view-change messages emitted by the server to create a valid view-change certificate as an iteration of a `ViewChangeListener`. Depending on whether or not the server has received any view-change for the current next view number or not, the view-change workflow may need to initialize the view-change certificate and view-change listener. This initialization process occurs on lines 8-17. After the view information is updated and the view-change certificate and view-change listener is initialized, the replica creates a view-change message and multicasts this over the PBFT network. Afterwards the view-change workflow needs to wait for the view-change listener to keep adding view-change messages until the view-change certificate becomes valid by having $2f + 1$ unique and valid view-change messages in its proof list. If this operation takes too long, then the timeout occurs, and the view-change workflow starts anew. This functionality is visible on lines 45-47, where the `listener` refers to a function that listens on a `Source<bool>` that only returns true whenever it receives an item on its reactive stream. The `Source<bool>` is only emitted to, by the server, when the view-change listener is finished making a valid view-change certificate. Once the view-change exchange is complete, the view-change workflow moves on to the new-view phase. This functionality is performed in the `ViewChangeProtocol` referred to in the next `WhenAny` function at line 56-58. If this operation takes too long, then the timeout once again is triggered, and the program starts at the top of the view-change workflow. The `ViewChangeProtocol` is responsible for having the new primary create a valid new-view message and multicast this message to the other replicas. The other replicas are responsible for validating that the information in the new-view message is valid. Finally, the workflow reprocesses the request that needs to be processed again. Once the view-change workflow shown in Listing 7.8 is finished, then the view-change is completed, and the application can once again start processing new client requests.

7.2 Workflow Details

```
1 ViewChange:
2 // Initialize
3 Serv.CurPrimary.NextPrimary();
4 Serv.CurView++;
5 ViewChangeCertificate vcc;
6 if (!Serv.ViewMessageRegister.ContainsKey(Serv.CurView))
7 {
8     vcc = new ViewChangeCertificate(Serv.CurPrimary, Serv.StableCheckpointsCertificate,
9         null, null);
10    Serv.ViewMessageRegister[Serv.CurView] = vcc;
11    ViewChangeListener vclListener = new ViewChangeListener(
12        Serv.CurView,
13        Quorum.CalculateFailureLimit(Serv.TotalReplicas),
14        Serv.CurPrimary,
15        Serv.Subjects.ViewChangeSubject,
16        false);
17    _ = vclListener.Listen(vcc, Serv.ServPubKeyRegister, Serv.EmitViewChange, null);
18 }
19 else
20 {
21     vcc = Serv.ViewMessageRegister[Serv.CurView];
22 }
23 var listener = ListenForViewChange();
24 var shutdownsource = new Source<bool>();
25 ViewChange vc;
26 CDictionary<int, ProtocolCertificate> preps;
27 if (Serv.StableCheckpointsCertificate == null)
28 {
29     preps = Serv.CollectPrepareCertificates(-1);
30     vc = new ViewChange(0, Serv.ServID, Serv.CurView, null, preps);
31 }
32 else
33 {
34     int stableseq = Serv.StableCheckpointsCertificate.LastSeqNr;
35     preps = Serv.CollectPrepareCertificates(stableseq);
36     vc = new ViewChange(stableseq, Serv.ServID, Serv.CurView,
37         Serv.StableCheckpointsCertificate, preps);
38 }
39 //View-change
40 Serv.SignMessage(vc, MessageType.ViewChange);
41 Serv.EmitViewChangeLocally(vc);
42 Serv.Multicast(vc.SerializeToBuffer(), MessageType.ViewChange);
43 CancellationTokenSource cancel = new CancellationTokenSource();
44 _ = TimeoutOps.AbortableProtocolTimeoutOperationCTask(shutdownsource, 10000,
45     cancel.Token);
46 bool vcs = await WhenAny<bool>.Of(
47     listener,
48     ListenForShutdown(shutdownsource)
49 );
50 if (!vcs) goto ViewChange;
51 cancel.Cancel();
52 //New-view.
53 Source<bool> shutdownsource2 = new Source<bool>();
54 CancellationTokenSource cancel2 = new CancellationTokenSource();
55 _ = TimeoutOps.AbortableProtocolTimeoutOperationCTask(shutdownsource2, 15000,
56     cancel2.Token);
57 bool val = await WhenAny<bool>.Of(
58     ViewChangeProtocol(preps, vcc),
59     ListenForShutdown(shutdownsource2)
60 );
61 if (!val) goto ViewChange;
62 cancel2.Cancel();
```

Listing 7.8: Overall source code for handling view-changes.

7.2.3.2.1 View-Change Listener Workflow Listing 7.9 shows the source code for our implementation of a view-change listener. Similar to the checkpoint listener, there are two separate ways to initialize an instance of the view-change listener. The first and most common method is for a timeout to occur in the protocol workflow due to not receiving the pre-prepare message. This, in turn, starts the view-change workflow for the replica, which initializes the view-change listener and view-change certificate for the following view number. The alternative way to start a view-change listener is for the server to receive a view-change message with a view number that it currently does not have in its view-change log. The server has a view-change log for each view-change certificate that the application is currently working on. This is for the case when the PBFT network is very large, and the replicas may disagree upon the next view number. Therefore, when the replica has been set to the inactive mode or participates for another next view number, it needs to collect any view-change messages for other next view numbers.

The view-change listener deviates a bit from the protocol workflow and checkpoint listener. The main difference is that it requires the ability to call upon a shutdown emit in the case where the system already has gotten $2f$ view-change messages. The reason for this functionality is due to making the system more efficient. The replica does not need to wait for a timeout to occur if it already has received $2f$ view-change messages since the PBFT network only requires that replica's view-change message to instantiate the new view. Therefore the process is sped up by calling a shutdown emit if it already has $2f$. Of course, this functionality is only helpful if the replica is still in active mode. This is the reason we added the option toggle on whether or not to use the shutdown emit functionality. To utilize the shutdown functionality, the boolean parameter `Shutdown` must be set to true, and the view-change listener must have a callback address to the server shutdown emit function. The server function schedules an item to be sent to the same `Source<bool> ShutdownSubject` that is used for the timeout functionality in the protocol workflow. This allows us to effectively stop the protocol workflow and set the application to be in inactive mode whenever the replica knows its vote can start a new view process.

The reactive listener performs relatively the same operators for the reactive stream as the normal protocol workflow did for its reactive handlers in the prepare and commit phase and like the checkpoint listener. Firstly, we want only to accept view-change messages that belong to the same view number used for the view-change certificate that the view-change listener is handling. Secondly, the view-change messages received are validated to make sure that it is a valid view-change message. Assuming the validation process is successful, the view-change message is added to the proof list of the view-change certificate. The final reactive operator validates the view-change certificate to see if it has received a sufficient number of unique and valid view-change messages in its proof list. After the view-change reactive listener is finished and a valid view-change certificate is ready, the callback function `finCallback` calls the server to emit a signal to the view-change workflow that the view-change certificate is finished. Once it receives the signal, the view-change workflow moves on to the next-view process of the view-change workflow.

As can be seen in Listing 7.9, there are two reactive chains used in the view-change listener. They both listen on the same `Source` object, but only one of them is active at

7.2 Workflow Details

a time. The first reactive chain belongs to the shutdown functionality and can be seen on lines 3-15. This one is used when the shutdown functionality is active for the view-change listener. The other reactive chain is seen on lines 17-26. They both perform the same initial **Where** clauses and the functionality for adding valid view-change messages to the view-change certificate. The only part differentiating between the two is the last **Where** clause. The shutdown reactive chain determines whether or not the view-change certificate has received $2f$ unique view-change messages in its proof list, while the other has the traditional check of $2f + 1$ unique view-change messages. Regardless of whether or not the shutdown functionality is used, the last reactive chain determines whether or not the view-change certificate is valid or not. Therefore, whenever the shutdown functionality has finished all of its operations, the program workflow naturally must wait in the second reactive chain for the view-change certificate to receive its last view-change message, which in likelihood should be replicas own view-change message.

```
1  if (Shutdown && shutdownCallback != null)
2  {
3      Console.WriteLine("With shutdown");
4      await ViewBridge
5          .Where(vc => vc.NextViewNr == NewViewNr)
6          .Where(vc => vc.Validate(keys[vc.ServID], ServerViewInfo.ViewNr))
7          .Scan(vcc.ProofList, (prooflist, message) =>
8              {
9                  prooflist.Add(message);
10                 return prooflist;
11             })
12         .Where(_ => vcc.ShutdownReached(FailureNr))
13         .Next();
14     Console.WriteLine("Calling shutdown");
15     shutdownCallback();
16 }
17 await ViewBridge
18     .Where(vc => vc.NextViewNr == NewViewNr)
19     .Where(vc => vc.Validate(keys[vc.ServID], ServerViewInfo.ViewNr))
20     .Scan(vcc.ProofList, (prooflist, message) =>
21         {
22             prooflist.Add(message);
23             return prooflist;
24         })
25     .Where(_ => vcc.ValidateCertificate(FailureNr))
26     .Next();
27 Console.WriteLine("Finished Listen view changes");
28 finCallback();
```

Listing 7.9: Source code for View-Change Listener

7.2.3.2.2 New-View Workflow The goal of this segment is to initialize the new common PBFT protocol state of the system after the view-change process is completed. The protocol state is determined by looking at the current stable checkpoint and the different protocol certificates obtained from the view-change messages. Thankfully the stable checkpoint can choose the last sequence number in which the majority of the replicas have agreed upon the application state. On the other hand, requests that are processed with higher sequence numbers do not have that guarantee. Therefore, the system is unsure whether or not the majority of replicas have actually managed to process these requests appropriately. The only way to be sure none of the protocol certificates

7.2 Workflow Details

are corrupted or incomplete is to redo their processing. Since each replica sends copies of their protocol certificates stored and their last checkpoint proof in their view-change message, it is possible to determine the requests that have to be reprocessed, including vital information needed to reprocess the request. The new primary is given the task to ready pre-prepare messages for each request that has to be reprocessed. In the unfortunate situation where there does not exist a protocol certificate record for a request that does need to be reprocessed, the digest of the request is set to `null`, indicating a missing operation. The list of pre-prepare phase messages created by the new primary is added to a new view message and multicasted to the other replicas in the PBFT. The new-view message also contains the view-change certificate created from the view-change message exchange. In this way, the new-view message is essentially a message to the other replicas in the network, relaying that it is the new primary of the PBFT system, and here is the proof to show it. The other replicas validate the new-view message they receive from the new primary. If the information in the new-view message is incorrect, the replica treats this as a failure of the new-view phase of the workflow and restarts the view-change process with the following view number. Otherwise, the replicas join the new primary in reprocessing the requests anew.

The source code for the reprocessing functionality can be seen in Listing 7.10. The code here is clearly very similar to the normal protocol workflow shown earlier in Listing 7.4 as we are literally attempting to do the same operations. The most obvious difference between the two workflows is the lack of pre-prepare phase for the redo processing since all pre-prepare phase messages have already been made and multicasted to the replicas. Since we know that the reprocess functionality is repeated until all of the pre-prepare phase messages for the protocol state are reprocessed, we decided it was best to iterate the reprocess functionality over the list of pre-prepare messages. As the pre-prepare phase messages have already been exchanged with the other replicas, the new primary only needs to listen and wait for the prepare phase messages from the other replicas during the prepare phase. Just like in the normal workflow, every replica, including the new primary, has to participate in the commit phase by creating a commit message and multicasting it over the PBFT network. The reactive operators used for the reactive chain are the same as those used in the prepare and commit phase in the normal protocol workflow. Although to avoid potentially cause issues with the Cleipnir execution engine, we decided to use a different `Source<PhaseMessage>` object for the reprocessing functionality. This `Source` object is known as *ReMesBridge* as can be seen on line 16 and 27. The reason why we decided to use different `Source` objects to differentiate between the normal workflow and the reprocess workflow is because we wanted to avoid potentially scheduling phase messages for the wrong workflow. From our experience scheduling for the same `Source` object for two separate workflows can, in the best case, be easily filtered out by the `Where` clauses. However, in the worst-case scenario, it can freeze the program due to scheduling an emit without any listeners for said `Source` object, when the program still needs to schedule other additional operations before the listeners are initialized. Therefore, to avoid this issue, we instead use two `Source<PhaseMessage>`, where the server schedules the phase message to be emitted to the appropriate `Source` object based on whether the application is active or inactive.

Unfortunately, just like the protocol workflow, the redo functionality can potentially fail and get stuck. On the other hand, this is not a significant issue as the view-change

7.2 Workflow Details

workflow can move on to the next view number and restart the view-change workflow if the reprocessing takes too long. This means the application never gets thoroughly stuck, even in the case the redo functionality fails. The redo functionality can fail if too many messages are received and emitted before the reactive listeners are ready. As the redo functionality immediately moves on to reprocess the next pre-prepare message whenever it is finished with another, it is possible to lose the phase messages for the next one if the other replicas are a lot faster. To mitigate the loss of phase messages, we decided to add wait periods where we know it is possible to miss phase messages. The first wait period is set to half a second and is performed right after the reactive listeners are defined on line 38. The other wait period is set to three-quarters of a second and is performed after the prepare phase is finished on line 53. Once the redo functionality has successfully reprocessed all of the pre-prepare phase messages for previous requests, the protocol logger should now have the two valid prepare certificates for each sequence number up to the point when the view-change was initiated. The view-change is then completed, and the application is once again set to active mode. We also perform garbage collection for the view-change log once the view-change is deemed successful, as seen in Listing 7.7.

7.2 Workflow Details

```
1 foreach (var prepre in oldpreList)
2 {
3     var precert = new ProtocolCertificate(
4         prepre.SeqNr,
5         prepre.ViewNr,
6         prepre.Digest,
7         CertType.Prepared, prepre
8     );
9     var comcert = new ProtocolCertificate(
10        prepre.SeqNr,
11        prepre.ViewNr,
12        prepre.Digest,
13        CertType.Committed
14    );
15    Serv.InitializeLog(prepre.SeqNr);
16    var preps = ReMesBridge
17        .Where(pm => pm.PhaseType == PMessageType.Prepare)
18        .Where(pm => pm.SeqNr == prepre.SeqNr)
19        .Where(pm => pm.ValidateRedo(Serv.ServPubKeyRegister[pm.ServID],
20            prepre.ViewNr))
21        .Scan(precert.ProofList, (prooflist, message) =>
22            {
23                prooflist.Add(message);
24                return prooflist;
25            })
26        .Where(_ => precert.ValidateCertificate(FailureNr))
27        .Next();
28    var coms = ReMesBridge
29        .Where(pm => pm.PhaseType == PMessageType.Commit)
30        .Where(pm => pm.SeqNr == comcert.SeqNr)
31        .Where(pm => pm.ValidateRedo(Serv.ServPubKeyRegister[pm.ServID],
32            prepre.ViewNr))
33        .Scan(comcert.ProofList, (prooflist, message) =>
34            {
35                prooflist.Add(message);
36                return prooflist;
37            })
38        .Where(_ => comcert.ValidateCertificate(FailureNr))
39        .Next();
40    await Sleep.Until(500);
41    if (!Serv.IsPrimary())
42    {
43        var prepare = new PhaseMessage(
44            Serv.ServID,
45            prepre.SeqNr,
46            prepre.ViewNr,
47            prepre.Digest,
48            PMessageType.Prepare
49        );
50        Serv.SignMessage(prepare, MessageType.PhaseMessage);
51        Serv.Multicast(prepare.SerializeToBuffer(), MessageType.PhaseMessage);
52        Serv.EmitRedisPhaseMessageLocally(prepare);
53    }
54    await preps;
55    await Sleep.Until(750);
56    Console.WriteLine("Prepare certificate: " + precert.SeqNr + " is finished");
57    Serv.AddProtocolCertificate(prepre.SeqNr, precert);
58
59    var commes = new PhaseMessage(
60        Serv.ServID,
61        prepre.SeqNr,
62        prepre.ViewNr,
63        prepre.Digest,
64        PMessageType.Commit
65    );
66    Serv.SignMessage(commes, MessageType.PhaseMessage);
67    Serv.Multicast(commes.SerializeToBuffer(), MessageType.PhaseMessage);
68    Serv.EmitRedisPhaseMessageLocally(commes);
69    await coms;
70
71    Console.WriteLine("Commit certificate: " + comcert.SeqNr + " is finished");
72    Serv.AddProtocolCertificate(prepre.SeqNr, comcert);
73 }
```

Listing 7.10: Redo Protocol Functionality

7.2.3.3 View-Change Evaluation

It was impossible to center the entire view-change functionality in the same function, just like with the checkpoint functionality. Although in the view-change functionality, a couple more issues needed to be handled. Just like in the checkpoint workflow, the view-change workflow could be initialized by any of the replicas. In addition, to managing timeout functionality and reprocessing functionality, the view-change workflow became a lot more complex than initially intended. We have done our best to divide up the jobs for the view-change workflow into separate segments based on the objective of the different jobs. We would argue that the view-change initialization process, view-change listener, and the redo protocol process all follow simple designs. On the other hand, the overall view-change functionality is more cluttered. Generally, this is because it has to both potentially initialize the view-change certificate and view-change listener. In addition to handling timeout functionality and restart functionality. The view-change functionality would be even more cluttered if the new-view process were not performed in its own `CTask`, even though the motivation for doing so was to keep it separate due to the timeout functionality.

Similar to the normal protocol workflow, all of the processes in the view-change functionality are run in `CTASK` asynchronous functions. Therefore, we cannot perform traditional asynchronous functionality inside the view-change functionality. However, we can still perform other asynchronous `CTASK` functions within the view-change workflow. The main benefit of asynchronous workflow for view-change functionality is to handle the restart functionality. By performing separate tasks in the view-change functionality in their individual asynchronous functions, we could take advantage of the `WhenAny` function to enable a timeout for each of the tasks. The `await` operator was used on the `WhenAny` function to wait for the resulting `CTask` regardless of whether or not the view-change functionality finished its operation or the timeout occurs first. The view-change functionality still takes advantage of the `await` operator to decide wait points for the asynchronous `CTASK` within the view-change workflow. However, combined with the timeout functionality, we know the functionality does not wait indefinitely for a result.

The view-change listener has very similar functionality to the checkpoint listener. It is run asynchronously and is not affected by normal protocol workflow or the view-change workflow. The result is also emitted in almost the same way. Unlike the checkpoint listener, the resulting view-change certificate is not directly emitted. Instead, a simple boolean value is emitted. In addition, the used `Source` object emitted to is also different. Unlike the checkpoint functionality, there is a deadline for an instance of checkpoint listener to finish its operation due to the timeout operations. Finally, the redo protocol process follows the same workflow as the original normal protocol workflow; therefore, it also takes advantage of `CTask`.

The reactive programming paradigm used for the view-change functionality follows both the normal protocol workflow and the checkpoint workflow. The reason being the parts of view-change functionality that uses the `Source` objects has practically the same workflow as either the normal protocol workflow or checkpoint workflow. The redo protocol processing workflow follows the normal protocol workflow, while the view-change listener functionality follows the checkpoint workflow. Therefore, since we already established

the benefits of reactive programming in the previous protocol segments, we will not do so again. On the other hand, view-change also uses reactive programming to stop the normal workflow when a view-change process starts. Although our current way of stopping iterations of the normal protocol workflow is functional, it is not very simplistic. This is mainly due to how challenging it is to interrupt a function with a waiting `Source` object to finish all of its operators. Our current solution is to use the `Merge` operator to interrupt the `Source` object if it receive an item from a separate `Source`. Although this workaround works for our pre-prepare reactive stream, it does have several problems. The first is that since we use the `Merge` operator to merge reactive streams, the other `Source` object used must listen for a matching object compared to the reactive stream. So in our case, we need a separate `Source<PhaseMessage>` for our `Merge` operator. However, this does cause issues in our prepare and commit reactive streams because of the changes in stream type after the `Scan` operator. Not to mention, the workflow must also account for the faulty item which was received by the `Source` object from the `Merge` operator. Like the checkpoint workflow, an earlier version of the view-change workflow existed that did not take advantage of the view-change listener for adding proofs to the view-change certificate. This original implementation also handled the view-change message validation, view-change certificate validation, and emit functionality inside a single append function. It did suffer from all of the issues we discussed in Section 7.2.2.4. In most cases, more problems occurred more frequently in the view-change certificate due to it requiring two callback functions instead of one. One to emit to the view-change workflow to move on and the other to call the stop functionality for the normal protocol workflow. Therefore, the change to using view-change listeners became a welcome addition.

7.3 Client

The client implementation created for the PBFT implementation is a primitive console application that is interactable by the user. The client uses interactivity to create unique operations that are to be handled by the PBFT algorithm. In our current PBFT implementation, we treat operations as simple string objects, meaning mostly any assigned string value can be used as an operation value. However, an exception to this rule is that the operation cannot contain a pipeline symbol. This is because the pipeline symbol is used as an end delimiter for serialized messages to resolve a TCP issue that can occur, which links two messages together. An operation is created by prompting the user for a value representing the value of the operation in the request message. Just like the replicas in the system, the client takes the network addresses stored in a JSON file and then establishes a socket connection to each of the network addresses. Unfortunately, this means the client can not be initialized before the replicas since it expects all replicas to be up and running when it attempts to establish socket connections.

In principle, the workflow for the client implementation is straightforward. The client starts by first initializing its connection to each of the replicas in the system based on the addresses found in the chosen JSON file. Then the user is prompted for the value to be used in the operation. Once the operation is deemed valid, the client creates a new request message using the operation provided by the user. The request is signed by

the client's private key and then multicast to the replicas in the PBFT network. After the client sends the request, it waits for the replicas to reply to the request the same way the normal protocol workflow does for a phase shift. A reply certificate is created, and the client uses a `Source<Reply>` to listen for reply messages reactively. When the `Source<Reply>` receives a new valid reply message, it is added to the reply certificate until the certificate has received at least $f + 1$ valid replies from different replicas. The $f + 1$ criteria is referred to as a weak certificate, which is a certificate that can guarantee that at least f non-faulty replica stored the request in its protocol log [38, p. 9], [39, p. 2]. Because the client is not part of the PBFT system, it only requires f number of replies to guarantee that the PBFT system properly processed the original request [30, p. 3], [38, p. 9].

If the reply certificate receives $f + 1$ replies from different replicas, the certificate is stored in the client's log. The client application restarts its workflow by again prompting the user for the next operation for the subsequent request. However, if the reply certificate does not become valid within a specific time duration, a timeout will occur, and the request is once again multicast to the PBFT network. This process is repeated until the $f + 1$ criteria is met. Unfortunately, if the PBFT application gets stuck on one of the client operations, the server does not accept the resent request as it believes it is already working on another request from the same client. Unfortunately, this usually leads to an endless loop. A way to get out of this loop would be for a view-change to occur on the PBFT. This is because the client status information on the replica is reset after the view-change is finished. The resent request is then treated as a new one, and the entire request processing starts anew.

The client shares a lot of the network-related code with the PBFT replicas. The main difference between the two lies in the client always being responsible for initiating the socket connection. The client also tries to reconnect to replicas it has previously been connected to but now is lost. The reconnection attempt is made whenever the client is about to multicast a request to the PBFT network. In the case where the reconnection fails, the client moves onto the other replicas. The client does, however, retry to reconnect to the lost replica whenever a new request is sent to the PBFT network.

We decided not to include persistency for the client implementation. Despite this, the network portion of the client still uses the Cleipnir execution engine when it emits incoming replies from the network layer to its reactive listener. The reason for this is because scheduling the emit using the Cleipnir execution engine enforces synchrony. Enforcing synchrony helps the client avoid a potential race condition that can potentially occur in this section of the code. We are currently uncertain in regards to what is causing this issue. We are running the reactive listener completely outside of Cleipnir's influence, which means additional threads are not supposed to be created. Despite this, we still have encountered race conditions in this section without using the Cleipnir execution engine.

Chapter 8

Discussion

In this section, we will summarize the benefits and difficulties we encountered when using the tools in our PBFT implementation. In addition to describing the benefits and disadvantages in regards to our chosen design.

8.1 Protocol Abstraction

Going into this thesis, our goal was to create protocol workflows that were both accurate and faithful to the original protocol description. Our approach to accomplish this was to keep all of the protocol workflows as simplistic by keeping the overall protocol workflow within a single function. The overall workflow should also perform each part of the protocol as synchronous as possible to resemble the protocol description. We refer to these principles as protocol abstraction. For our PBFT implementation, we only managed to keep all of the protocol-related source code for the normal protocol workflow within a single function. Both the checkpoint and view-change workflow had to be split into several code segments based on the jobs they performed. We would say that we were relatively successful for the normal protocol workflow in terms of simplicity. This is primarily due to the source code performs each protocol phase while still being decently readable. Unfortunately, this came at the cost of not being able to handle pre-prepare protocol messages out of order.

Both the checkpoint and view-change workflows were separated into several code segments primarily due to how both processes could be instantiated in by any of the replicas in the PBFT, making it severely challenging to keep the workflows orderly and somewhat synchronous. It was generally more natural to segment the checkpoint workflow because it is entirely independent of the normal protocol iterations. This is especially notable for the garbage collection section of the process. Keeping the garbage collection section together with the checkpoint listener would have been relatively messier than having an active checkpoint listener send the result to the server which is responsible for performing the garbage collection process. As for the view-change workflow, it follows a more orderly workflow similar to the normal protocol workflow. However, the view-change workflow

required support for restarting processes should the new primary turn out to be faulty. By splitting the operations in the workflow, it would be a lot simpler to provide the timeout functionality needed. Not to mention the view-change functionality itself is quite long, requiring all of the code to be written in a single function would not have simplified the workflow. The workflow would instead be rather messy. In short, we can conclude that there is a somewhat of a fine balance between keeping code orderly and simplistic and providing the functionality we desire. The more complicated the functionality is, the more likely it is that the code also becomes somewhat harder to understand. Therefore, consensus algorithms are very likely to become complicated because the functionality they require is usually quite complex.

8.2 Asynchronous workflow

Asynchronous programming has, in most situations, been beneficial to our PBFT implementation. Note that when we refer to asynchronous programming in this section, we do include both `Task` and `CTask` asynchronous operations. Both normal `Task` and Cleipnir `CTask` operations have practically the same workflow. They both take advantage of the `async/await` workflow, and they both follow the TAP abstraction. Because of this, it makes it easy for a developer to create a `CTask` workflow if they are familiar with the traditional asynchronous workflow. The `await` operator has been valuable for delegating wait points for both the asynchronous workflows and Cleipnir reactive workflow. The use of the `await` operator allowed us to create an abstraction for our asynchronous workflow to be read like a synchronous workflow, which greatly helped readability for the workflows. Since the protocol workflows needed to be run together with Cleipnir, we needed to use `CTask` asynchronous programming. Except for not being able to use any other traditional asynchronous operations within any of the protocol workflows, there were no other significant disadvantages to running the protocol workflow with Cleipnir asynchronous operations. The `CTask` still allowed for the protocol workflow to be run asynchronously, allowing for an easy way to create both several instances of the protocol workflows but also have them run independently. We would argue this is a minor drawback in comparison to support both persistency and the use of Cleipnir's reactive framework for our protocol workflows. Traditional asynchronous programming has not been helpful for the protocol workflow. On the other hand, traditional asynchronous programming has been a big cornerstone to the application network layer. Meaning traditional asynchronous programming works well with both server handling and with socket programming.

The problems in mixing the two types of asynchronous operations together have already been mentioned numerous times throughout this thesis. As a result, to avoid causing any race conditions or inconsistent states for the application, the application had to be designed to separate the asynchronous operations from each other as much as possible. However, this does create additional limitations for a developer if they were to design a consensus algorithm with the Cleipnir framework. Although asynchronous operations do seem quite handy, it is essential to avoid using them when it is not necessary, or the application could be slowed down as a result. Combine this last issue with combining both `CTask` and `Task` is rather disastrous.

8.3 Usage of Cleipnir

In this section, we discuss the benefits provided by the Cleipnir framework. We are primarily focusing on Cleipnir's reactive framework and Cleipnir persistency functionality

8.3.1 Reactive Operators

Cleipnir reactive framework has been beneficial for dealing with operations related to protocol messages. There are three contributions provided to the workflow for using the reactive workflow to handle the protocol messages. The first is that all of the message-related operations, such as validation steps, are performed in sequence, meaning the ordering for the operations is clear and that operators are not called unless the previous operator succeeds. The second reason is that it segments all of the code related to the handling protocol messages away from the overall protocol workflow, making the overall workflow easier to understand. The final reason is that the ideal design of the protocol message handler is an event-based process due to the program not knowing when the protocol message is received, meaning reactive programming is the best approach to use. The overall workflow for Cleipnir reactive workflow is simple in design. We require only a single `Source` object to be initialized with a specific object type and have this be shared between two parts of the system. The first part of the system emits the item, while the other part listens to the stream for any incoming items and performs the desired reactive operators on the received item. The usage of reactive stream operators is generally straightforward but may be difficult for developers unfamiliar with query languaging. The Cleipnir reactive framework has, for the majority of the time, had the reactive operators we needed, despite having a low amount of reactive operators compared to other reactive API. Regardless, Stidsborg has demonstrated that the process of adding other reactive operators from the Reactive X library is not challenging. The `Merge` operator was added to Cleipnir reactive framework during this project due to our need to stop the normal protocol workflow, which took less than a day. The source code for the reactive workflow has also shown to be quite reusable, considering we have used relatively the same workflow for all of our protocol workflows. From our usage of the Cleipnir reactive framework for our implementation of the PBFT workflows, we would argue that it is well suited for event-driven programming.

A major drawback we have encountered with the Cleipnir reactive framework is that the Cleipnir execution engine is almost always is required to be used when performing an emit to the protocol workflow. This particular issue may only be relevant for our design. Still, it is pretty clear that if any object is to be transformed by the reactive stream, the Cleipnir execution engine is needed to schedule the emit of the protocol message. Otherwise, the reactive stream attempts to handle multiple protocol messages concurrently, which causes the object to become inconsistent. With the help of the Cleipnir execution engine, we can enforce a FCFS ordering on the protocol messages, avoiding this issue entirely, but that in itself may not always be beneficial. The most notable issue is when an emit takes place within a `CTask` asynchronous operation. Scheduling operations and performing the desired operations using the Cleipnir execution is treated as a traditional

8.3 Usage of Cleipnir

asynchronous operation. Therefore the workflow within the `CTask` cannot wait for the scheduled operation to finish as it is performed on another thread, which can cause problems if the scheduled operations are vital for the rest of the workflow.

We can also document that stopping an active workflow that is waiting for the result of a reactive stream is not very simple. We discussed in the Section 7.2.3.1 how we made a workaround for this issue, but it is quite the situational solution. From our experience, it is more beneficial to design any workflows with more restrictive `Source` operators early, to keep the workflow active but essentially filter out any of the future protocol messages as soon as possible. It does waste resources, but the overall design of workflow will be a lot more desirable.

The most detrimental disadvantage we encountered regarding the Cleipnir reactive workflow lies in trouble dealing with protocol message messages being received out of order. We already discussed how our current PBFT implementation struggles in certain parts of the program to handle protocol messages being received out of order. This issue does have two known workarounds, although both do limit the design of the workflow. The first workaround is to initialize and have each `Source` object start listening on the reactive streams as early as possible in the workflow. Otherwise, any protocol messages received out of order or received too early are lost. This is the workflow we primarily used in our workflows, unfortunately not always possible to use this workaround, as shown in our pre-prepare phase implementation. The other known workaround would be to use unique `Source` objects for each and every protocol message type to avoid any of the other `Source` object filtering out any messages received for that protocol type. However, this workaround needs a lot of extra `Source` objects to be initialized and individual object types for every type for every protocol message, which can become quite extensive.

To summarize, Cleipnir's reactive framework is well suited for segmenting operations in a consensus algorithm that is reliant on event-based programming. In addition, it is relatively easy for developers to use once the basics have been learned. It also helps make the general workflow more compact and readable thanks to its ability to use chain operators on the reactive stream. However, the reactive framework can struggle with handling several events types when they are all listening on the same `Source` object when the ordering of the events matters. The reactive framework is also tricky to use on a workflow that requires handling forms of exceptions to normal workflow. Since consensus algorithms must be able to handle situations where parties in the network stop responding, this can become a frequent issue. However, there do exist workarounds for handling some of the issues discussed. We believe that better workarounds are also bound to be discovered in the future.

8.3.2 Persistency

Persistency has not been the primary focus for our PBFT implementation. We have, however, been attempting to take advantage of the Cleipnir persistency functionality to persist relevant data for our implementation. Our main contributions here lie in making sure all the protocol workflows are run within `CTask` asynchronous operations. In ad-

8.3 Usage of Cleipnir

dition, we used Cleipnir hybrid persistency functionality to create appropriate serializer and deserializer for our protocol objects. Unfortunately, the current PBFT implementation does not support functional persistency. The reason why the persistency does not work properly is uncertain. We have discovered at least two issues. The first issue is that the protocol logger, for some reason, does not persist all the entries in the logger. As of now, no distinguishable pattern has been found with the data that gets lost. Either way, losing certificates in the logger does create big problems for our restarted application. The other problem is that some of the `Source` objects which are linked to the server gets duplicated when the system is rebooted. This means that in the persisted system, there suddenly exist two `Source` objects with the exact same reference. This is a big problem because each time the server emits a message to the duplicated `Source` object, it emits the message to both the original and duplicate `Source` objects. This, in turn, can cause two identical iterations of the protocol workflow to occur, meaning they work on the same sequence number. In the worst-case scenario, both protocol iterations store the resulting protocol certificates on the same sequence number, leading to scenarios where the logger suddenly has four certificates recorded for a single sequence number. The logger is only meant to store a maximum of two protocol certificates for a protocol iteration, so this is quite the issue for future protocol states.

However, we have tested the persistency functionality for our implementation by using smaller practical examples and unit tests. We can verify that the persistency works well for smaller segments of our application, such as testing the persistency for most of our protocol objects. We can at least certify that the hybrid persistency functionality works quite well for Cleipnir. It is both convenient and straightforward to choose segments of an object to persist using the object-oriented design to the serializer and deserializer. Although, a developer can potentially make mistakes if they unaware of the limitations to Cleipnir ability to persist certain data types.

Chapter 9

Conclusion

This chapter concludes the thesis by first listing the lessons we learned while working on the thesis. Then we list the potential future work which can be applied to the PBFT implementation. Finally, a conclusion is drawn for the work performed for this thesis.

9.1 Lessons Learned

9.1.1 Consensus algorithms

At the start of this thesis, our knowledge in regards to consensus algorithms was limited to having previously implemented the Paxos algorithm using Golang language [52]. We had never encountered any information in regards to the PBFT consensus algorithm; therefore, some time was spent on learning the inner workings of the PBFT algorithm. In addition, Cleipnir had already been used to implement the Paxos and Raft consensus algorithms. Therefore some time was also spent on understanding the basics of the Raft consensus algorithm to help understand the source code used for the Raft implementation. We realize that transition from one consensus algorithm to another when looking solely at the protocol descriptions is not all that complicated. Many components for dealing with specific issues regarding consensus algorithms are shared for many consensus algorithms. As a result, it became simpler for someone familiar with one consensus algorithm to understand another.

9.1.2 Asynchronous Programming

Asynchronous programming proved to be suitable for the network layer of the application. In addition, asynchronous programming became an excellent boon for designing a multi-client application when used on the protocol workflows. Since the `CTask` also took advantage of the `async/await` workflow, implementing `CTask` functions was just as simple

9.1 Lessons Learned

as the .NET traditional asynchronous workflow.

However, we acknowledge that our inexperience with the background operations occurring in the `async/await` workflow hurt our initial design for our application. This, combined with our wrong assumption regarding `CTask`, delayed our thesis considerably.

9.1.3 Reactive Programming

At the beginning of this project, we had very little to no previous experience in regards to reactive programming. Therefore it was challenging to learn the basics of reactive programming. The most significant complication became understanding Cleipnir's reactive functionality knowing only the basics of reactive programming. Using the Cleipnir reactive functionality is straightforward once you learn the basics. However, making a direct comparison to the official reactive documentation [11] to Cleipnir.Rx was not simple. This primarily due to the cornerstones having different name schemes between the two.

Regardless, we have demonstrated in our application that we ascertained the knowledge to use the reactive framework to handle the PBFT protocol messages and their resulting operation. It was showed that the framework was appropriate for handling event-driven scenarios in consensus algorithms. In addition, simply enough to have our reactive workflow be reused for several parts of our reactive implementations.

9.1.4 Cleipnir

Starting the thesis, we had little to no prior experience with working with the Cleipnir framework. The most challenging part of learning how to use Cleipnir functionality was the lack of a detailed documentation. We only had access to its source code and a couple of well written practical examples. Bakkevig previously also seemed to struggle in this department in his thesis [3, p. 43-44]. This is somewhat our fault as we were not accustomed to learning about frameworks by reading their source code. During our study, practically all tools and frameworks used had a form of written documentation. However, not all frameworks have well-written documentation. Although, most commonly used frameworks usually have some form of community that uses which you can discuss unexpected issues when the need arises. As Cleipnir is still in development, it does not have a large community. We also understand that since Cleipnir is constantly updated, writing a detailed documentation could be seen as wasteful because the functionality changes frequently. This means that the documentation would also need to be continuously updated, leading to a lot of extra work for each update. However, we do share Bakkevig's opinion that if Cleipnir is to become well liked by developers, time must be invested into writing at least a small description for its unique components as well as a guideline for how each tool available in the framework should be used and what users should actively avoid. Stidsborg was available to answer any questions we had regarding Cleipnir. Despite this, flawed assumptions were made, leading to problems for the development of our implementation. Safe to say, we learned that making assumptions

can be quite detrimental, and we should have perhaps queried Stidsborg earlier in the development about the difference between Cleipnir implementation of commonly used classes compared to their traditional counterparts.

9.2 Future Work

As mentioned in Chapter 6 our current cryptographic signature architecture is susceptible to impersonation and spoofing attacks. Clearly, keeping public keys ephemeral and generating them uniquely before start-up was not a smart design when the system supports persistency. Creating static private and public keys is also not recommended since this design would make the system less secure. One solution would be to generate a couple of unique key pairs for each replica and have these stored securely or given to the system by a separate trusted system. This system could, for instance, be a database where the cryptographic keys are stored encrypted. During system start-up or during certain scenarios, such as view-changes and or system restarts, the replica reassigns its current cryptographic key pairs and re-establishes its secret key with the other replicas in the system. The other replicas only accept the renewed connection if the separate system acknowledges that the public key given matches one of the unique public keys listed for that replica.

We are currently using a digital signatures scheme for all message types, except for the session messages. This is frankly unnecessary and only slows down the system. The desired alternative is to follow the original PBFT system model and use MAC for authentication instead, as this would be more efficient. Although, we still recommend continuing to use the digital signature structure for view-change and new-view messages. Otherwise, the view-change workflow would need to be redesigned to follow the more advanced workflow described in Castro's and Liskov's updated paper for PBFT [38, p. 410-414].

The protocol workflow currently suffers from the inability to handle pre-prepare and prepare being received out of order. In addition, prepare messages can also be lost if the message is received before the prepare listener is initialized. As described in Section 7.2 this issue can cause the workflow to become stuck if too many prepare messages are lost while the workflow waits for a pre-prepare message. This is something that should be corrected if the application is to be used in the future. A workaround to this problem would be to have a timeout functionality active when the workflow waits for the desired number of prepare and commit messages. The timeout is stopped if both the reactive listeners have successfully created both protocol certificates. Otherwise, the timeout expires, and the reactive listeners are terminated using the same functionality used for the pre-prepare listener. For this functionality to be possible, another `Source` object would need to be added to the workflow to work with the `Merge` operator. This is because the reactive stream for reactive listeners to the prepare and commit message is of type `Stream<CList<PhaseMessage>>` due to the stream being transformed by the `Scan` operator.

Solving the actual message ordering issue is a lot more complicated. It is not as simple

9.2 Future Work

as initializing the prepare listener earlier, as the listener needs to filter away any phase message with a different sequence number than its current iteration. Unfortunately, non-primary replicas set the current sequence number based on the received pre-prepare message, creating quite the conundrum. A solution to this problem is making the server store copies of the phase messages received in the network layer. By having this logger store a list of phase messages received for a sequence number within a dictionary, it would be possible for the workflow to easily search for missing phase messages. The phase message records stored in this logger would have to be garbage collected once the protocol has successfully created the two desired protocol certificates for the given sequence number. However, this would cause additional complexity to the protocol workflow as functionality for looking up, and re-emitting lost phase messages would need to be added.

Currently, our application does not fully support persistency. In the future, it would be favorable for both Cleipnir and our application if the issues described in Chapter 7 can be fixed to allow for our application to thoroughly test Cleipnir's capability in regards to persistency. The groundwork has been laid for the application to work with persistency. This includes assigning all protocol object types their proper serialization and deserialization functions for Cleipnir to use, which have been tested on a smaller scale and works as intended. In addition, the network functionality for replicas to reconnect to the system has already been implemented and tested. The only thing left is for the system to successfully read the data stored by Cleipnir's storage engine and successfully restore its old state. There are at least two notable issues that must be fixed for the application to become persistent. The first issue is that the original `Source` objects are duplicated by having Cleipnir somehow restore the original `Source` while also creating the desired new copy, which was supposed to replace the old. Currently, both `Source` objects react whenever new items are emitted to them by the network layer, meaning that for the protocol workflow, two iterations are created for a single sequence number. This, in turn, creates issues for the logger when multiple records for the same sequence number are stored. The second issue is that the logger synchronization isn't working properly and as a result, records in the logger disappear after the replica restarts. This issue likely due to the synchronization not being fully finished before moving with other operations, or the synchronization is not done correctly, and as a result, some records are skipped. We assume this issue is caused by incorrect usage of `Sync` points set for Cleipnir, resulting in the state not being persisted correctly. As for the duplicate `Source` objects, we are frankly not quite sure how this issue occurs. We theorize that it may occur due to some records being persisted in multiple objects, and because of this, when the objects are persisted, the objects are not treated as the same `Source` object, leading to the duplicate `Source` object. If this is the case, the issue lies in the relationship between the server and the protocol workflow.

Generally, the application functionality could be a lot more advanced than it is now. Currently, the only operation the application performs after a request is processed successfully through the PBFT algorithm is simply printing the message attached to the request to the console window. The message is then added to a `CList` representing the state of the system. In the future, it would be beneficial if the application functionality was changed to be a bit more practical. For instance, changing the message content in the request to be an operation that is to be performed by the application instead of a string. The state list would then rather store a record of the operation performed and

whether or not the application successfully performed the requested operation. In order to change the application functionality, the client functionality for creating requests must also be adjusted.

9.3 Conclusion

In conclusion, we achieved our goal of creating a simplistic PBFT implementation using Cleipnir with the intended focus of making it faithful to the protocol description, which also is designed to take advantage of asynchronous and reactive programming paradigms. The result is PBFT implementation that can perform the PBFT protocol over several multiple clients and has functional checkpoint and view-change functionality. We managed to design a normal workflow that fit our original criteria, but unfortunately, the protocol struggles with handling out-of-order protocol messages. The checkpoint and view-change workflow became too complex for the processes to be handled within a single function. Persistency functionality was sadly not successful for our PBFT implementation. Asynchronous programming is shown to be helpful when designing consensus algorithms. Asynchronous programming was notably useful in regards to networking functionality and for designing multi-client protocol workflows. Similarly, reactive programming turns out to be fairly helpful for handling the operations regarding protocol messages and other event-based processes. Reactive programming, however, did appear to struggle with protocol message ordering when using synchronous design. These two programming paradigms showed quite clearly that they work well together. We believe implementation consensus algorithms can be further simplified using these tools in the future, despite the problems addressed in this thesis. In regards to the Cleipnir framework, we acknowledge that the overall workflow of the Cleipnir reactive framework is user-friendly and has, for the most part, the functionality desired for designing a proper event handler for a consensus algorithm. We were unsuccessful in evaluating Cleipnir's persistency functionality on our application. However, based on our experience with using the hybrid persistency functionality on our implementation. In addition to testing the persistency functionality for smaller parts of the program, we deem Cleipnir's persistency functionality to be excellent. To conclude this thesis, we do believe that the tools we have tested and evaluated during our PBFT implementation do make it easier to design consensus algorithms. In the future, we believe that consensus algorithms can be implemented simpler and more accurately to the protocol description. However, we acknowledge that due to the complex nature of distributed systems, it will be challenging to create accurate consensus algorithm implementations due to the numerous problems that can occur.

List of Figures

4.3.1 Practical Byzantine Fault Tolerance Normal Workflow	21
4.5.1 Practical Byzantine Fault Tolerance View-Change	24
6.1.1 Overall architecture of the PBFT implementation network	28
6.3.1 Summary of the file architecture for the PBFT implementation	33
6.4.1 Application divided into persistent parts and ephermeral parts and how they interact	36

Listings

2.1	Example of async/await workflow	7
3.1	Object Store example	12
3.2	Execution engine example	13
3.3	Example of chaining Cleipnir reactive operators	13
3.4	Source object example	14
3.5	Object persistency initializer	15
3.6	Serialize/Deserialize code example	16
3.7	Example of a CTask function	17
6.1	Example of server and protocol interaction using Cleipnir scheduler	36
7.1	Code section for the request handler	40
7.2	Source code for pre-prepare phase for primary replica	42
7.3	Source code for Pre-prepare phase for non-primary replica	43
7.4	Source code for Prepare and Commit phase	48
7.5	Source code for the Checkpoint Listener	51
7.6	Reactive handler for new stable checkpoints	52
7.7	Handling timeout for the normal protocol workflow and initiate the View-Change process	56
7.8	Overall source code for handling view-changes.	58
7.9	Source code for View-Change Listener	60
7.10	Redo Protocol Functionality	63

Appendix A

PBFT Implementation Source Code

- The PBFT implementation can be found on this [Github repository](#).
- The source code used for the replica implementation is found in this [sub directory](#).
- The source code used for our client implementation is found in this [sub directory](#). The unit tests can unfortunately not be all run concurrently, due to some issue with the network tests. We therefore recommend running each folder separately, and re-run the test that fail uniquely once they fail.
- The Github repository provides a detailed explanation in regards to how to run the PBFT implementation both locally and with docker containers.

Bibliography

- [1] T. Inc. (2021). ‘Distributed system,’ [Online]. Available: <https://www.techopedia.com/definition/18909/distributed-system> (visited on 18/03/2021).
- [2] M. van Steen and A. S. Tanenbaum, *Distributed Systems Third edition Preliminary version 3.01pre*. Maarten van Steen, Feb. 2017, pp. 456–458, NOTE:Book was previously published by: Pearson Education, Inc, ISBN: 978-90-815406-2-9.
- [3] E. Bakkevig, ‘Implementing a distributed key-value store using corums,’ Jun. 2020, pp. 7–52. [Online]. Available: <https://hdl.handle.net/11250/2679782>.
- [4] vic. (Aug. 2019). ‘From distributed consensus algorithms to the blockchain consensus mechanism,’ [Online]. Available: https://www.alibabacloud.com/blog/from-distributed-consensus-algorithms-to-the-blockchain-consensus-mechanism_595315 (visited on 03/03/2021).
- [5] T. S. Sylvest, H. Meling and L. Jehl, ‘Cleipnir - framework support for fault-tolerant distributed systems,’ Nov. 2020, pp. 1–18.
- [6] T. S. Sylvest. (2021). ‘Cleipnir = persistent programming in .net.’ NOTE:Github repository private for Stidsborg ask permission in order to get access., [Online]. Available: <https://github.com/stidsborg/Cleipnir.PersistentProgramming> (visited on 21/03/2021).
- [7] Lupu2 and stidsborg. (Jul. 2021). ‘Practical byzantine fault tolerance implementation,’ [Online]. Available: <https://github.com/Lupu2/PBFT-Master/tree/main> (visited on 12/07/2021).
- [8] BillWagner, Varad25, n1c, mrlife, IEvangelist, Youssef1313, nschonni, dkreider, pkulikov, Thraka, damabe, kendrahavens, nextn, DennisLee-DennisLee, mikkellbu, nemrism, mairaw, mikeblome, mjhoffman65, guardrex and tompratt-AQ. (Apr. 2020). ‘Asynchronous programming with async and await,’ [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/> (visited on 31/01/2021).
- [9] karelz, nschonni, kotx, TimShererWithAquent, nemrism, nextn, mairaw, BillWagner, Shivien, jzeferino, Mikejo5000, mjhoffman65, PeterSmithRedmond, guardrex, tompratt-AQ and yishengjin1413. (Mar. 2017). ‘Asynchronous server socket example,’ [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/framework/network-programming/asynchronous-server-socket-example> (visited on 10/06/2021).

- [10] R. Weeks. (May 2020). ‘C# sockets programming - ep01.’ Youtube Video, [Online]. Available: <https://www.youtube.com/watch?v=rrlRydqJbv0> (visited on 10/06/2021).
- [11] ReactiveX. (2021). ‘Reactivex,’ [Online]. Available: <http://reactivex.io/> (visited on 12/03/2021).
- [12] R. Stropek. (Oct. 2020). ‘C# async programming - part 1: Conceptual background.’ Youtube Video, [Online]. Available: <https://www.youtube.com/watch?v=FIZVKteEFyk> (visited on 25/01/2021).
- [13] AramT. (2018). ‘Your ultimate async / await tutorial in c#.’ NOTE:Tutorial spans 7 webpages, the link will redirect to the introduction page., [Online]. Available: <https://www.codingame.com/playgrounds/4240/your-ultimate-async-await-tutorial-in-c/> (visited on 09/03/2021).
- [14] BillWagner, DCtheGeek, IEvangelist, TimShererWithAquent, gewarren, Thraka, Youshef1313, nschonni, pkulikov, nemrism, sguitardude, ryanliang88, nxtn and mairaw. (Aug. 2020). ‘Task asynchronous programming model,’ [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/task-asynchronous-programming-model> (visited on 31/01/2021).
- [15] E. Agafonov, *Multithreading in C# 5.0 Cookbook*. Packt Publishing, Limited, Nov. 2013, pp. 190–232, ISBN: 978-1-84969-764-4. [Online]. Available: <https://ebookcentral.proquest.com/lib/uisbib/detail.action?docID=1572912>.
- [16] N. Parente. (Sep. 2020). ‘.net async programming in a nutshell,’ [Online]. Available: <https://nelsonparente.medium.com/net-async-programming-in-a-nutshell-dc01c2e71a20> (visited on 04/07/2021).
- [17] E. Zamora-Gómez, P. García-López and R. Mondéjar, ‘Continuation complexity: A callback hell for distributed systems,’ in *Euro-Par 2015: Parallel Processing Workshops*, S. Hunold, A. Costan, D. Giménez, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer and M. Alexander, Eds., Cham: Springer International Publishing, 2015, pp. 286–298, ISBN: 978-3-319-27308-2.
- [18] K. Patel. (Dec. 2016). ‘What is reactive programming?’ [Online]. Available: <https://medium.com/@kevalpatel2106/what-is-reactive-programming-da37c1611382> (visited on 12/03/2021).
- [19] J. Liberty, *Programming Reactive Extensions and LINQ*, eng, 1st ed. 2011., ser. Expert’s voice in .NET. Berkeley, CA: Apress : Imprint: Apress, 2011, ISBN: 1-280-39231-2.
- [20] dotnetsheff. (Aug. 2020). ‘The reactive extensions for .net.’ Youtube Video, [Online]. Available: https://youtu.be/6yjl_h7-WYA (visited on 31/01/2021).
- [21] MuleSoft. (2021). ‘What is an api? (application programming interface),’ [Online]. Available: <https://www.mulesoft.com/resources/api/what-is-an-api> (visited on 27/06/2021).
- [22] . Foundation. (Nov. 2020). ‘Reactive extensions,’ [Online]. Available: <https://github.com/dotnet/reactive> (visited on 31/01/2021).
- [23] ReactiveX. (2021). ‘Observable,’ [Online]. Available: <http://reactivex.io/documentation/observable.html> (visited on 12/03/2021).

- [24] M. Developer. (Nov. 2018). ‘Reactive extensions for .net developers.’ Youtube Video, [Online]. Available: <https://youtu.be/c9Yq-XE58hA> (visited on 31/01/2021).
- [25] ReactiveX. (2021). ‘Subject,’ [Online]. Available: <http://reactivex.io/documentation/subject.html> (visited on 12/03/2021).
- [26] Mozilla and individual contributors. (Mar. 2021). ‘Concurrency model and the event loop,’ [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop> (visited on 22/03/2021).
- [27] G. for Geeks, Shivi_Aggarwal, ukasp and 29AjayKumar and Rajput-Ji. (Feb. 2021). ‘Program for fcfs cpu scheduling set 1,’ [Online]. Available: <https://www.geeksforgeeks.org/program-for-fcfs-cpu-scheduling-set-1/> (visited on 22/03/2021).
- [28] Microsoft. (2021). ‘Ting du kommer til å like med sql server 2019,’ [Online]. Available: <https://www.microsoft.com/en-us/sql-server/sql-server-2019> (visited on 22/03/2021).
- [29] ReactiveX. (2021). ‘Introduction,’ [Online]. Available: <http://reactivex.io/documentation/operators.html> (visited on 16/03/2021).
- [30] M. Castro and B. Liskov, ‘Practical byzantine fault tolerance,’ 1999, pp. 1–14. [Online]. Available: <http://pmg.csail.mit.edu/papers/osdi99.pdf>.
- [31] P. Hooda. (Dec. 2019). ‘Practical byzantine fault tolerance(pbft),’ [Online]. Available: <https://www.geeksforgeeks.org/practical-byzantine-fault-tolerancepbft/> (visited on 12/01/2021).
- [32] H. Dang, Z. Gao, I. J. amd L. Luu and D. Sivasankaran, ‘Practical byzantine fault tolerance,’ Slide Presentation, Mar. 2016. [Online]. Available: <https://www.comp.nus.edu.sg/~rahul/allfiles/cs6234-16-pbft.pdf>.
- [33] Xangle. (Oct. 2018). ‘What is practical byzantine fault tolerance (pbft)?’ Youtube Video, [Online]. Available: <https://www.youtube.com/watch?v=M4RW6GAwryc> (visited on 12/01/2021).
- [34] Binance.com. (Jan. 2021). ‘Byzantine fault tolerance explained,’ [Online]. Available: <https://academy.binance.com/en/articles/byzantine-fault-tolerance-explained> (visited on 03/03/2021).
- [35] K. Khullar. (Aug. 2019). ‘Implementing pbft in blockchain,’ [Online]. Available: <https://medium.com/coinmonks/implementing-pbft-in-blockchain-12368c6c9548> (visited on 12/01/2021).
- [36] L. Lamport, R. Shostak and M. Pease, ‘The byzantine generals problem,’ *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982, ISSN: 0164-0925. DOI: [10.1145/357172.357176](https://doi.org/10.1145/357172.357176). [Online]. Available: <https://doi.org/10.1145/357172.357176>.
- [37] W. Zhao, *Building Dependable Distributed Systems*. John Wiley & Sons, Incorporated, Mar. 2014, pp. 239–284, ISBN: 9781118912706.
- [38] M. Castro and B. Liskov, ‘Practical byzantine fault tolerance and proactive recovery,’ in *ACM Transactions on Computer Systems*, 2002, pp. 399–416. [Online]. Available: <https://doi.org/10.1145/571637.571640>.
- [39] X. Hao, L. Yu, L. Zhiqiang, L. Zhen and G. Dawu, ‘Dynamic practical byzantine fault tolerance,’ in *2018 IEEE Conference on Communications and Network Security (CNS)*, 2018, pp. 1–8. DOI: [10.1109/CNS.2018.8433150](https://doi.org/10.1109/CNS.2018.8433150).

BIBLIOGRAPHY

- [40] S. Nolan. (Nov. 2018). ‘Pbft — understanding the consensus algorithm,’ [Online]. Available: <https://medium.com/coinmonks/pbft-understanding-the-algorithm-b7a7869650ae> (visited on 08/01/2021).
- [41] _kitchen. (Nov. 2019). ‘Consensus series: Pbft,’ [Online]. Available: <https://medium.com/thundercore/consensus-series-pbft-3e011e7f3691> (visited on 08/01/2021).
- [42] I. Bitwise IO. (2018). ‘Pbft architecture,’ [Online]. Available: <https://sawtooth.hyperledger.org/docs/pbft/releases/latest/architecture.html#pbft-operation> (visited on 17/03/2021).
- [43] H. Meling, J. I. Olsen, T. E. Lea and L. Jehl. (Apr. 2021). ‘Gorums,’ [Online]. Available: <https://github.com/relab/gorums> (visited on 14/03/2021).
- [44] R. Anderson, K. Larkin and M. Wasson. (Apr. 2021). ‘Tutorial: Create a web api with asp.net core,’ [Online]. Available: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-5.0&tabs=visual-studio> (visited on 14/03/2021).
- [45] (2021). ‘What is a spoofing attack?’ [Online]. Available: <https://www.malwarebytes.com/spoofing/> (visited on 24/05/2021).
- [46] P. Fox. (2020). ‘Transmission control protocol (tcp),’ [Online]. Available: <https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:transporting-packets/a/transmission-control-protocol--tcp> (visited on 08/06/2021).
- [47] Newtonsoft. (2021). ‘Json.net popular high-performance json framework for .net,’ [Online]. Available: <https://www.newtonsoft.com/json> (visited on 05/05/2021).
- [48] w3Schools. (2021). ‘C# enum,’ [Online]. Available: https://www.w3schools.com/cs/cs_enums.asp (visited on 04/05/2021).
- [49] Google. (2021). ‘Channels,’ [Online]. Available: <https://tour.golang.org/concurrency/2> (visited on 06/05/2021).
- [50] A. Stefanuk. (Mar. 2020). ‘What is sql? a beginner’s guide to the sql language,’ [Online]. Available: <https://learntocodewith.me/posts/sql-guide/> (visited on 27/06/2021).
- [51] BillWagner, MightyPen, mairaw, nxtn, nemrism, mjhoffman65, mikeblome, guardrex and tompratt-AQ. (2015). ‘Goto (c# reference),’ [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/goto> (visited on 11/05/2021).
- [52] Google. (2021). ‘Go,’ [Online]. Available: <https://golang.org/> (visited on 03/06/2021).