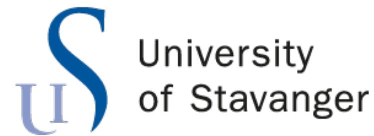
Faculty of Science and Technology

MASTER'S THESIS

Study program/Specialization: Computer Science - Reliable and Secure Systems	Spring semester, 2021 Open
Writers: Sander Tunge Aspøy Helene Larsen	……………………………………… (Writer's signature) ……………………………………… (Writer's signature)
Faculty supervisor: Leander Jehl External supervisor(s):	
Thesis title: Diversification for HotStuff through WebAssembly	
Credits (ECTS): 30	
Key words: • BFT • WebAssembly • HotStuff • Browsers • Blockchain • Diversity • Reliability • Consensus • State Machine Replication • WebRTC • Peer-to-Peer	Pages: 104 + enclosure: link to GitHub repository Stavanger, 15. June 2021 Date/year



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

Diversification for HotStuff through WebAssembly

Master's Thesis in Computer Science
by

Sander Tunge Aspøy

Helene Larsen

Supervisor

Leander Jehl

June 15, 2021

"The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do."

Theodor Holm Nelson

Abstract

By design, the goal of Byzantine Fault Tolerant (BFT) protocols is to protect against malicious or malfunctioning nodes. A BFT protocol in itself is only as secure as the system it is running on. In a perfect world, this would be enough. However, in the real world, multiple layers of security are crucial. Our goal is to expand on the reliability and security provided by existing BFT protocols through diversification with WebAssembly. WasmStuff is a WebAssembly compatible BFT protocol based on relab/hotstuff's [1] implementation of the *HotStuff* [2] protocol. Consequently, to our knowledge, we have created the first complete browser-based BFT protocol. We looked into different ways of working with WebAssembly, such as using different compilers and running a WebAssembly module inside or outside a browser. Furthermore, we have explored different ways of creating network connections with regards to WebAssembly, and the solution we opted for is described in detail. The networking provided in this thesis is a peer-to-peer connection compatible for use with WebAssembly. Moreover, we designed a runtime for WasmStuff that can be modified to work with other BFT protocols.

Our evaluation shows that WasmStuff performs similarly to relab/hotstuff when it runs on Windows without WebAssembly, which indicates that our modifications for WebAssembly did not impact the performance by much. We met our goal of diversity through the use of WebAssembly as WasmStuff runs in the major browsers. WasmStuff has the ability to run both cross-browser and cross-platform, providing even more diversity.

Acknowledgements

We want to express our sincere gratitude to our supervisor Leander Jehl. Through the weekly meetings and discussions, he has helped us overcome obstacles and has given us essential insight on what to focus on when it comes to diversification and WebAssembly.

Contents

Abstract v

Acknowledgements vii

Abbreviations xi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Definition . . . 2
- 1.3 Contributions . . . 3
- 1.4 Outline . . . 3

2 Background 5
- 2.1 Byzantine Fault Tolerance . . . 5
  - 2.1.1 Origin . . . 6
- 2.2 The HotStuff protocol . . . 6
- 2.3 Lazarus: Automatic Management of Diversity in BFT Systems . . . 9

3 WebAssembly 11
- 3.1 Definition . . . 11
- 3.2 Use cases . . . 12
- 3.3 WebAssembly and Go . . . 13
- 3.4 WebAssembly System Interface . . . 13
- 3.5 Using WebAssembly for BFT systems . . . 14
- 3.6 Challenges . . . 15
  - 3.6.1 Obstacles . . . 15
  - 3.6.2 Desired features . . . 16

4 Research and Analysis 17
- 4.1 Existing Approaches of WebAssembly and BFT protocols . . . 17
  - 4.1.1 WebAssembly in browser . . . 17
  - 4.1.2 BFT in browser . . . 19
- 4.2 Analysis . . . 20
  - 4.2.1 Requirements . . . 20

4.2.2 Networking with WebAssembly 21
4.2.3 Execution environment 22

5 Design and Implementation **25**
5.1 Design overview 25
5.2 Web Server 26
5.3 Website 28
5.4 WebRTC for WebAssembly 29
5.5 WasmStuff 31
5.5.1 WasmStuff Interface 31
5.5.2 Runtime 34
5.5.3 Consensus protocol 39
5.6 BFT SMR Controlled Chess 40

6 Experimental Evaluation **43**
6.1 Experimental Setup 43
6.1.1 Performance testing 43
6.2 Experimental Results 44
6.2.1 Diversification analysis 51

7 Discussion **53**

8 Future Work **57**
8.1 Security Features 57
8.2 General Improvements 58
8.3 Further Diversification 59

9 Conclusion **61**

List of Figures **61**

List of Tables **65**

A User Guide **67**

B Marshalling and Unmarshalling **71**
B.1 Marshalling 71
B.2 Unmarshalling 72

C Raw Benchmark Data **75**

Bibliography **89**

Abbreviations

AOT Ahead-Of-Time

API Application Programming Interface

BFT Byzantine Fault Tolerant

CDN Content Delivery Network

CVSS Common Vulnerability Scoring System

DOM Document Object Model

FIFO First In, First Out

gRPC gRPC Remote Procedure Call

ICE Interactive Connectivity Establishment

IP Internet Protocol

JS JavaScript

LTU Local Trusted Unit

NaCl Native Client

NAT Network Address Translation

OpenCV Open Source Computer Vision Library

OS Operating System

PC Partial Certificate

QC Quorum Certificate

relab Resilient Systems Lab

RPC Remote Procedure Call

SDP Session Description Protocol

SMR State Machine Replication

STUN Session Traversal Utilities for NAT

UI User Interface

VM Virtual Machine

WASI WebAssembly System Interface

Wasm WebAssembly

WebRTC Web Real-Time Communication

WS WebSocket

Chapter 1

Introduction

1.1 Motivation

Hearing the words Byzantine Fault Tolerant (BFT) protocol, distributed systems likely comes to mind. It is one of many techniques used to ensure reliability and security in distributed programming. The name BFT stems from the Byzantine Generals problem. The Byzantine Generals need to inform their armies about the next action to take. The problem is how can they be sure that the armies act on the same orders. If one of the generals is malicious, the whole operation can fall to pieces. A BFT protocol is based on this concept and has steps and rules that ensure that the system makes progress and that all participating parts receive the correct distributed decision. Following that, only as long as more than $\frac{2}{3}$ of the replicas are correct, the system is said to operate correctly and thus move forward and make progress. These protocols prevent malicious actions of protocol participants and tolerate arbitrary software failures and outsider attacks on the system. While this sounds very great in theory, it is pretty easy to miss the hidden weakness. Different systems require different instruction sets, requiring different software implementations. Developing a single application is much work, often leading to choosing uniform hardware and software, resulting in shared vulnerabilities. Suddenly a weakness found on a single machine could compromise a majority of the system.

Our goal is to strengthen the BFT protocol by diversification. The WebAssembly (Wasm) standard is an excellent tool in this quest. It is featuring a standardised instruction set for code execution on different hardware. By harnessing the power of Wasm, we are motivated to make a BFT system that is easy and available for use by anyone. Secure distributed systems should not have to be run exclusively on high-end server architectures in data centres. We believe that by using Wasm, it will be possible to create a single program that can be compiled and ran on almost all available hardware.

Wasm has already made quite the waves in the web development communities by bringing power to the browser. The ability to run powerful applications client-side in the users' browser can save significant expenses for a developer or company. For many small developers, server cost can be a hurdle many will not even try to cross. With Wasm the cost for powerful server hardware can be alleviated, leading to great new ideas and tools. The Wasm expansion called WebAssembly System Interface (WASI) is making great progress on bringing Wasm out of the Web. The solution we present is a solution focused on execution in browsers. With the usage of browsers, our hardware pool is vastly diversified, strengthening the system against shared hardware exploits. With the future development of both WASI and other Wasm compilers, our developed source code can gain diversification and resilience on the software level as well.

1.2 Problem Definition

The problem description given for this thesis was the following:

"The goal of this project is to utilise WebAssembly technology to provide a diverse execution environment for a BFT algorithm. To reach this goal, it is planned to build on an existing implementation of the HotStuff algorithm developed at UiS [1].

The goal of this thesis is to compile the existing algorithm to WebAssembly and try to run the resulting system using different Virtual Machines (VMs) or Interpreters.

The long term goal is to develop a runtime that can be used with different BFT algorithms. If time permits, you may be able to identify, how to separate the runtime from the BFT algorithm."

With the help of Wasm, we can diversify BFT protocols and thus increase the existing tolerance in BFT protocols. The diversification comes from being able to run the same software on different underlying systems through the use of Wasm.

Wasm code can either be run in a browser or a Wasm supported VM. As of today, all major browsers support Wasm, giving us diversification if a system of different browsers run the HotStuff protocol. The same goes for using different VMs to run the HotStuff protocol. Unfortunately, the technology for running Wasm using different VMs lacks support for a lot of key features in HotStuff, and thus the main focus of this thesis will be browsers. The technology needed to implement the aforementioned solution is very experimental. Thus this thesis will include research of different possibilities to be able to meet the requirements of both the protocol and of Wasm.

1.3 Contributions

In this thesis, we make the following contributions:

- we research the current possibilities of implementing a BFT protocol with Wasm. The research covers the basic requirements for any given BFT protocol as well as Hotstuff specific requirements.
- we present different approaches of adapting BFT protocols to be able to compile to Wasm. Some of the approaches are currently not feasible but may be useful once the technology has improved.
- we propose WasmStuff; an adaptation of the relab/hotstuff protocol which can compile to a .wasm file that can be run in a web browser using Web Real-Time Communication (WebRTC).
- we propose a runtime for WasmStuff that can be modified to work with other BFT protocols using Wasm.
- we evaluate the performance of WasmStuff in different test environments.
- we examine the diversification achieved by WasmStuff.

1.4 Outline

The outline of the remaining part of this thesis is as the following:

Chapter 2. A must-read if you have never heard of BFT, or a suggested read if you want to know more about the specific BFT protocol used in this thesis and about a similar project we are comparing our results with.
Chapter 3. Can be skipped if you are familiar with Wasm; otherwise, this chapter will introduce you to it.
Chapter 4. Useful chapter if you are planning to implement your own Wasm solution, as it contains information about what is feasible and not in Wasm when it comes to networking and other BFT related requirements.
Chapters 5.- 7. The main event, this is where our proposed solution is described along with the experimental results.
Chapter 8. If you wish to expand on our work, we have written some suggestions for you in this chapter.
Chapter 9. Skip straight to this chapter if all you want to know is if we were successful in implementing a BFT protocol in the browser, as this shortly summarises it for you.

Chapter 2

Background

In this chapter, we introduce the BFT concept and why it was needed. In addition, we explain the different phases of *HotStuff* and give a short summary of the Lazarus project.

2.1 Byzantine Fault Tolerance

The objective of BFT protocols is to endure arbitrary (i.e., Byzantine) failures of replicas while still taking actions critical to the system's operation. A system in this context would, for example, be a State Machine Replication (SMR). A Byzantine faulty replica could misbehave in many different ways, such as distributing the wrong transaction or not distributing a transaction at all. In the case of the replica sending the wrong transaction but distributing that same wrong transaction to the whole system, the BFT protocol will consider this as correct. This is due to BFT protocols not handling the correctness of the transaction itself.

A BFT SMR protocol ensures that non-faulty replicas agree on the order of execution for transactions, despite the efforts of f Byzantine replicas. This, in turn, ensures that the n-f correct replicas will run the transactions identically and so produce the same response for each transaction. Due to the fact that BFT protocols do not handle the correctness of a transaction itself, if enough correct replicas, in a BFT SMR system, agree on the wrong transaction, the transaction would still be executed [2].

2.1.1 Origin

The BFT concept is based on the Byzantine Generals' Problem from 1982. This problem is a logical dilemma about a group of Byzantine generals who may have issues with agreeing on their next move. The issues come from the lack of trust with each other and the means of communication. In this scenario, each general has their own army, and they are based at different locations around the targeted city of attack.

The generals can either attack or retreat. Whichever command they choose does not matter, but they have to agree on what they will do. The requirements are the following:

- Each general has to decide whether to attack or retreat.
- A decision is final, i.e. it cannot be changed.
- All generals have to agree on a command and execute it in a synchronised manner.

The generals will send a courier bringing their chosen command to the others to come to an agreement. This is where the problems arise, as the messages can get delayed, destroyed or lost. Despite a successful delivery of a message, it does not mean all failures were avoided. One or more generals could have decided to send out one command and then do the complete opposite to confuse the other generals, which would lead to a total failure [3].

Deducted from the Byzantine Generals' Problem, a system is considered Byzantine fault tolerant if all correct replicas agree. In practice, in a BFT system with $N = 3f + 1$ replicas, at **most** f replicas **may** be Byzantine faulty.

2.2 The HotStuff protocol

In 2018, HotStuff was proposed: a BFT SMR protocol [2]. In HotStuff, all replicas know of all the participating replicas, as well as their public keys. The rest of this section includes a description of the different phases of HotStuff, as described in [2], and how the relab/hotstuff differs from [2].

HotStuff is a leader based BFT replication protocol, where the replicas decide on and execute client commands. A full round of HotStuff includes all the phases and messages it takes from a proposal has been made until the command has been executed. Rounds in HotStuff are referred to as views, meaning every proposal is identifiable by a view number. Every replica holds information about the current view number, the Quorum Certificate (QC) with the highest view number known to the replica (qc_{high}), and the last committed block b_{lock}. The b_{lock} is used to ensure that only blocks descending from a confirmed block are signed.

The protocol consists of four phases: prepare, pre-commit, commit and decide. In the prepare phase, the leader proposes a new command. The command will either come from a command buffer or, if the buffer is empty, it generates a new command. The leader then creates a new block with this command, and creates a QC and adds its signature to it. After adding its own signature, the leader sends the block to all the other replicas and waits for a quorum of responses. A quorum is the least amount of responses needed for the leader to perform the next operation and eventually execute the command. Once the other replicas receive the block, they check if the block is a descendant of the current b_{lock} and if the view in the qc_{high} of the block is higher than the view of the current b_{lock}. If so, they sign the block and create a Partial Certificate (PC) and sends it back to the leader.

The leader will verify the PCs using the public key of the replica who sent the PC. When the leader receives a quorum of PCs from the replicas, the block has entered the pre-commit phase. Here the leader verifies the signatures and adds the PCs to the QC, creating a pre-commit QC. The leader then sends this new QC to all the replicas and waits for their responses. The replicas receive the new QC, showing that a quorum was achieved, create a new PC and send it back to the leader.

The block has now entered the commit phase, and the leader creates a new QC, called a commit QC, and sends it to the replicas and waits for their responses. At this stage, when the replicas receive a commit QC, they become locked on the current proposal block and updates the b_{lock} to be the current block; $b_{lock} = b$. The replicas then create another PC and send it back to the leader just as before.

At last, the block has reached the decide phase, and the leader creates a final decide QC. The leader then considers the b_{lock} proposal to be committed and executes the command in the proposal. As the final step, the leader sends the decide QC to the other replicas. When the replicas receive the decide QC they also consider the b_{lock} proposal to be committed and execute the command in the proposal.

Looking at the different phases, it is clear that many of them are very similar, thus chained HotStuff was made. Through pipelining, excess messages are eliminated by using the prepare messages of new proposals as the different phase messages of previous proposals. However, this causes a proposals command not to be executed until three more proposals have been made. The pipelining is illustrated in Figure 2.1.

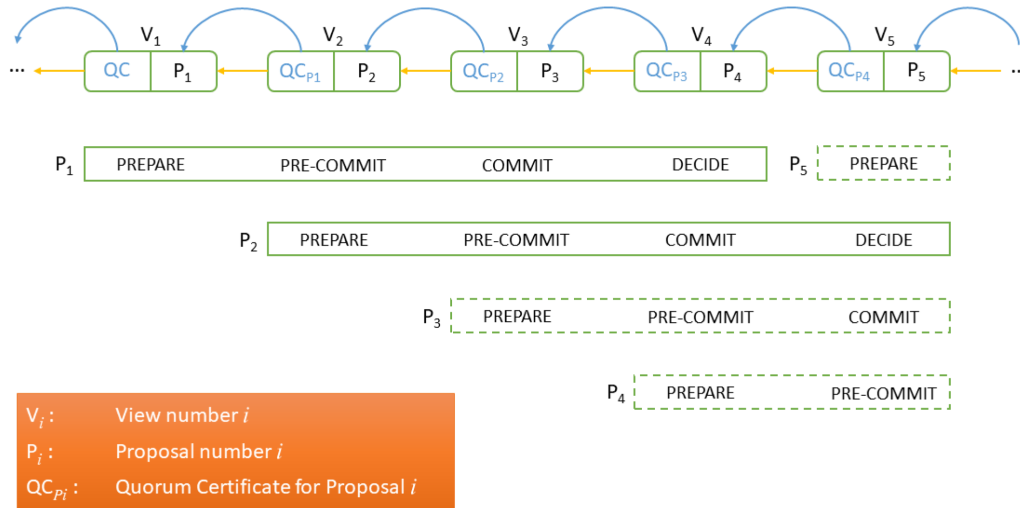


Figure 2.1: Illustration of how pipelining in HotStuff is done.

In this thesis, we are using the Resilient Systems Lab (relab) implementation of HotStuff [1], from now on referred to as relab/hotstuff, as our BFT protocol. The relab/hotstuff implementation is based on [2], but they had to make some adjustments to the protocol as they were implementing it using Gorums Remote Procedure Calls (RPCs). Gorums is a gRPC Remote Procedure Call (gRPC) wrapper that uses code generation to produce an RPC library that can be used to invoke quorum calls. These quorum calls are the same as general RPCs, but only a quorum of the responses are handled [4].

In order for us to use the relab/hotstuff, implementation with Wasm, some adjustments had to be made. Thus, we propose WasmStuff, an implementation of the relab/hotstuff protocol with modifications to make it suitable for compilation to Wasm. The adjustments and modifications are explained in Chapter 5, along with a detailed description of why and how we did this.

2.3 Lazarus: Automatic Management of Diversity in BFT Systems

Lazarus [5] manages the deployment and execution of diverse replicas in BFT systems. By monitoring the current vulnerabilities of the replicas in the system, Lazarus can employ a metric to measure the risk of having common weaknesses in the set of replicas. Lazarus' objective is to achieve diversity in replica sets used in BFT systems. Most BFT systems assume diversity without having any supporting mechanisms to back up said assumption. Monitoring the systems replicas and their current vulnerabilities, Lazarus can measure the risk of having common weaknesses in the replica set. To ensure diversity, Lazarus will reconfigure the set of replica that is found vulnerable. This way, the system will run the "most diverse" set of replicas. This paper presents a method for assessing the risks a group of replicas may have and evaluate said method. They have also included an extensive evaluation of their prototype, which shows that it is feasible to run BFT systems with diversity for specific configurations.

By collecting data on new vulnerabilities and exploits from open source security feeds, they can create an up-to-date and evolving database of vulnerable devices. After identifying common vulnerabilities, they calculate and assign a severity score. A configuration risk is calculated for a set to represent how vulnerable it is to simultaneous compromisable attacks. Their last presented method used in Lazarus is the risk assessment calculation for an already deployed replica set. If the risk crosses a defined threshold, a replacement process will be started. Replicas presenting the highest risk get taken out of the running configuration, and a replacement is selected. Removed replicas are placed in quarantine until they receive the needed security patches.

The complete Lazarus implementation consists of two planes with four modules. In the control plane, the managers reside. The data manager is responsible for gathering and interpreting vulnerability data. The risk manager uses the gathered data from the data manager to assess the risk of every replica. The deploy manager performs automated mobilisation and demobilisation of replicas. Lazarus uses virtualisation with VirtualBox [6] to deploy the application when preparing replicas. Each replica has a Local Trusted Unit (LTU) installed, which is shielded from the internet. The Lazarus controller controls the LTU to ensure safe and synchronous behaviour.

The execution plane is where the replicas are deployed. It is layered, having a system configuration at the bottom, containing the replicas. The replica's stack is configured in the following way:

- Service
- BFT-Library
- Java Virtual Machine
- Operating System (OS)

The Lazarus system was tested using a four replica BFT system. The configuration pool included 17 different OS versions. The Lazarus controller was trained with four years of vulnerability data before being tested on eight months of data. System vulnerability was compared with four other diversification strategies. These other strategies were: equal, random, common and Common Vulnerability Scoring System (CVSS) v3. A short description of these strategies is provided below:

Equal: Randomly selects an OS used for all replicas

Random: All replicas use a random OS; one replica is replaced each day randomly

Common: Minimised shared vulnerabilities between chosen OS's

CVSS v3: Similar to Lazarus, testing combinations to lower CVSS v3 score

The results presented in the Lazarus paper indicates that Lazarus was the strategy that performed best each of the eight months the testing included. Results were presented as a percentage value of compromised system runs per month. Overall, Lazarus had an average of around 18% compromised runs throughout the eight-month period. The CVSS v3 strategy performed closest to Lazarus with an average of around 86% compromised runs. Considering the strategy closest to Lazarus had almost five times more compromised runs, it is reasonable to say they achieved their goal of managing diversity in BFT systems.

Chapter 3

WebAssembly

This chapter explains what Wasm is and when to use it. The compatibility of Go with Wasm is also described, along with an introduction to the WebAssembly System Interface (WASI).

3.1 Definition

"WebAssembly (abbreviated Wasm*) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications."* [7]

The quote above is the definition given by the developers of Wasm. In other words, Wasm is designed to enable LLVM-supported [8] languages to run on web pages. Many different compilers can compile source code written in an LLVM-supported language to the Wasm format and run it in the same sandbox as JavaScript (JS) code, such as TinyGo [9]. The goal of Wasm is to be able to execute at native speed, which would give greater freedom compared to websites running on plain JS.

The developers of Wasm made five high-level goals [10] for what they wanted Wasm to be. In essence, they wanted to make a binary format to serve as a compilation target, which would be portable, size- and load-time-efficient. By taking advantage of common hardware capabilities on numerous platforms, the binary format would be able to be compiled to execute at native speed. They also wanted to design Wasm to execute within and integrate well with the existing Web platform, such as the following: allowing synchronous calls to and from JS access browser functionality through the same Web Application Programming Interface (API)s that are accessible to JS defining a human-editable text format that one can convert to and from the binary format.

Today, the Wasm developers have reached some of their goals, but there is still much work to do to meet all of them. After their release, they specified that they are not aiming at replacing JS, but rather for Wasm to be a complement to it [11]. JS will remain the single, privileged dynamic language of the Web, despite the fact that Wasm can compile multiple languages for execution on the Web. It is expected that Wasm and JS will be used in numerous configurations together. Examples of such could be as whole native compiled apps that leverage JS to glue things together, or apps consisting of mainly HTML/CSS/JS with a couple of high-performance Wasm modules.

There has also been some scepticism around creating a new standard when asm.js already exists [11]. To that, the developers of Wasm highlight two main benefits provided by Wasm. The first benefit is the time it takes to decode the binary format of Wasm. Experiments indicate that the decoding of the Wasm binary format is 20 times faster than the time it takes to parse JS. The second benefit is how much easier it is to add the features required to reach native levels of performance using Wasm compared to using simultaneous asm.js constraints of Ahead-Of-Time (AOT)-compilability. Even though developing a new standard introduces new costs, for Wasm the cost is comparable to a large new JS feature instead of a fundamental extension to the browser model. This statement comes from the ability of a browser to implement Wasm directly inside its existing JS engine. Based on this, they conclude that the benefits outweigh the costs.

3.2 Use cases

On the Wasm website, they have a long list of different use cases [12], which are applications they think would benefit from Wasm and which they also keep in mind when continuing the design of Wasm. The list is divided into three categories: inside the browser, outside the browser and how Wasm can be used. For browsers, some of the things they mention are games, music applications, remote desktops and virtual/augmented reality with very low latency. As of today, many of the things on the list have been made possible in Wasm, but due to non-frequent updates on the website, it is hard to find out precisely what has been done and what still remains. Inside the browser, the developers wished to include the possibility of peer-to-peer applications, both decentralised and centralised. This is particularly interesting for our thesis as we need peer-to-peer connections between the replicas. This is also a good example of one of the things on the list that has been handled. To our knowledge, WebRTC is the only possible way to achieve peer-to-peer connectivity as of today. The reason for this is explained further in Section 5.4.

3.3 WebAssembly and Go

We have found two ways of compiling code from Go to Wasm: Go's built-in compiler and TinyGo [9]. TinyGo will not only compile the code to Wasm but will make the .wasm file significantly smaller as well. By design TinyGo was made to compile Go code for use on microcontrollers [13], as the Go language was not intended for this use. TinyGo only support parts of the Go language at the moment. The project is in constant development, and the support is constantly under improvement. Due to the main focus of microcontroller targets, the compiler creates small and efficient binaries. These small binaries are also a desired feature for web use to decrease load times for websites.

The TinyGo compiler will optimise memory usage and create the smallest possible binaries. In comparison, the Go compiler focuses on performance. One of the reasons for the smaller file sizes from TinyGo is that they use their own runtime, which is smaller than the one used by Go. Just as a quick test, we built the smallest possible "hello world" project. The absolute minimum file size of the resulting Wasm file from Go was 1254 KB, while TinyGo reduced the file size to only 575 bytes. For the Go compiler, the size significantly increases when importing packages. Importing and using the print formatting package *fmt* to print the same message increases the Wasm file size to 2126 KB.

Unfortunately, we have not been able to use the TinyGo compiler for this project. The problem with using TinyGo is that there are still some standard packages that are not supported yet. Most notably, TinyGo has not yet been able to make the *net* package compile. Additionally, the cryptography package is not fully supported yet. While it is possible to work around these restrictions by extracting this from Wasm and perform these actions in JS instead, we wish to keep as much of our implementation within Wasm as possible.

3.4 WebAssembly System Interface

As the name suggests, Wasm was designed to run well on the Web. However, as it does not make any assumptions that are web-specific and does not provide web-specific features, it is possible to employ Wasm in environments other than on the Web. Wasm uses APIs exclusively to interact with the outside world. When running on the Web, it uses the existing Web APIs provided by the browser. If it were to run outside of the Web, there is no standard set of APIs to write Wasm programs to. Thus the WebAssembly System Interface (WASI) was developed.

WASI has a clean set of APIs that does not depend on browser functionality and can be implemented on multiple platforms by multiple engines. Furthermore, WASI focuses on system-oriented APIs, covering files, networking and with more to come in the future. These APIs do not depend on browser functionality but are still capable of being run in a browser.

The core design of WASI follows the concept of capability-based security [14]. Capability-based security is, in essence, a concept relying on having a capability to access an object. In this context, a capability is defined to be a communicable, unforgeable token of authority. The capability refers to a value that references an object and a set of access rights associated with said object. UNIX-like file descriptors identify files, directories, network sockets and other resources. External tables use these as indices for the different capabilities. Furthermore, WASI provides no ability to access the outside world without an associated capability, which is similar to how core Wasm provides no ability to access the outside world without calling on imported functions.

To give an example, when a user wants to access a file with WASI, they use an openat-like [15] system call instead of a typical open system call. The difference between an open system call and the openat system call is that openat interprets the pathname relative to the directory referred to by the file descriptor rather than relative to the current working directory of the calling process. The openat-like system call requires a file descriptor for a directory that holds the file, representing the capability to open files within that directory. With WASI it is also possible to grant capabilities for directories on launch, and the WASI libc [16] will maintain a mapping from their file system to the file descriptor index, which represents the needed capability. When a program calls open, it will then look up the file name in the map, and if it exists, it will automatically supply the relevant directory capability [17].

3.5 Using WebAssembly for BFT systems

As of today, to use Wasm for BFT systems is a bit tricky. This is because Wasm is not intended to be used to host servers (as of today), meaning the support for essential features needed in a BFT system are missing. That said, the idea of doing this is great. It would strengthen the idea of a distributed system to encompass the software. In BFT systems today, the main idea is that different hardware runs the same software to minimise the number of replicas failing at once due to either attacks or a fault in the hardware itself. Adding to that by also using different software to run the program, it is fair to expect even fewer replicas to fail. To our knowledge, WasmStuff is the first BFT protocol fully implemented using Wasm.

3.6 Challenges

In this section, we introduce the different challenges that using Wasm introduces and a list of features that we hope to see implemented in Wasm shortly.

3.6.1 Obstacles

While Wasm brings many great improvements to the world of web programming, it is also in its early stages. Working with the first release of a language or a standard often proves both exciting and frustrating. Wasm is no different. The lack of conventional networking support was the first challenge we encountered. Building a BFT SMR, a typical server-to-server application, on a platform designed for client applications seems almost out of reach. The lack of socket listener support forces us to think differently from before.

Browsers are designed for simple client applications and typically rely on servers to perform demanding computations or actions. With Wasm, near-native execution speeds remove the need to rely on a server for computationally demanding actions. Due to the single-threaded design of a browsers execution environment, it is still not best practice to run demanding operations on a website. This resulted in web workers being developed.

Web workers allow JS scripts to be executed on a site running in the background. From a JS file, we can create a web worker. The main browser window can communicate with the web worker through the postMessage() function. This keeps the main window ready and available to perform both simple user actions and User Interface (UI) updates while computations happen in the background. A web worker is a helpful tool for Wasm developers. However, there is currently a significant restriction that bars us from using web workers. The WebRTC API is not available for use on web workers. Working around this would mean deploying the networking outside of Wasm, and that is undesirable.

During troubleshooting and testing of our implementation, we ran into performance issues where due to resource starvation, the UI and browser window stopped updating. Few seconds after that, the whole window goes white and is unresponsive. From earlier experience, we suspected that this was caused by a resource-intensive execution in the Wasm code or an infinite loop. Through extensive code review and troubleshooting, we discovered that when we made our code more effective, we achieved a higher throughput. However, this resulted in the window crashing even earlier.

When troubleshooting, we used several print statements to log the progress. From Wasm a print would be executed through a syscall to JS console to log the message. The more instructions WasmStuff managed to execute, the more calls were made, which again resulted in the window crashing faster. A single execution of the print call is not very resource-intensive, but resource usage quickly becomes a problem when executed rapidly in a series.

3.6.2 Desired features

Since Wasm is a new and developing standard, it is fair to expect that some features are lacking. These are some of the features we desire. First and foremost, the ability to listen for and accept incoming network connections would be a game-changer for developers. With the current state of browsers, it is unlikely that we will see socket listeners in Wasm in the near future.

WASI is making exciting progress for multiple origin languages, and full networking capabilities seems to be on the horizon. Hopefully, soon we will see improved availability of WASI. Upon fulfilment of their goals, WASI stands to deliver excellent diversification potential. We also wish to see WASI implement full networking support. Currently, there is a lack of support for file system support in Wasm, and even though this is supported in WASI, we hope this is something Wasm will support someday soon.

While not a feature in Wasm, we wish to see full support for multithreading in the browser; or, more realistically, at least extended availability for web workers. While web workers provide a way to use multithreading, it is far from the simplicity and usability that, for example, a Go application would allow.

Chapter 4

Research and Analysis

Before delving into the development of WasmStuff, we will examine and analyse existing implementations of BFT protocols and projects made in Wasm. Using the knowledge gained from examination and analysis, we will determine the requirements for our implementation. When researching, we tried out different approaches to ensure it would be possible to meet our goal. Many of these approaches failed, but we include them in this chapter to illustrate how challenging and frustrating it can be to work with new technology.

4.1 Existing Approaches of WebAssembly and BFT protocols

This section introduces some existing approaches of Wasm projects and BFT protocol implementations in browsers. The reason for including both of these is that Wasm is a relatively new addition to the development toolbox; thus, we see ourselves forced to expand our view from pure Wasm BFT implementations to general Wasm and BFT implementations.

4.1.1 WebAssembly in browser

Many developers are excited about the possible performance enhancements that Wasm is bringing. Many exciting projects are using Wasm in the browser to run computationally demanding code [18]. Some examples are Open Source Computer Vision Library (OpenCV), which has been ported to Wasm to allow for better performance when used in the browser, and ssheasy [19], which is a Wasm implementation of an SSH terminal. The ssheasy project allows a user to SSH into a computer directly from the browser.

Another big project that has taken advantage of the Wasm technology is Google Earth [20]. Google Earth is a project initially written mainly in C++, and it was intended as an installable desktop application. As smartphones became more widely used, they were able to port it to both Android and iOS, using NDK [21] and Objective-C++ [22] to retain most of their C++ codebase. In 2017 they used Native Client (NaCl) to port the C++ code to be able to run Google Earth on the web. The downside was that NaCl was only compatible with the Google Chrome browser. With WebAssembly, they could finally make it available on almost all of the major browsers.

Despite it now being available on almost all major browsers, such as Firefox, Edge and Opera, there is a difference in the state of support for the different browsers. In the newest version of Edge, Google Earth runs as smoothly as it does in Google Chrome because Edge is now Chromium-based. Chromium is an open-source codebase for browsers, meaning browsers based on this shares a lot of the same code and functionalities.

In both Chrome and Edge, the Wasm support is strong. One of the key features supported for Wasm in both of these browsers is multithreading. Multithreading is especially important for Google Earth, as it is constantly streaming data to the browser, making it resource intensive. Thanks to multithreading, all this work can be done in the background on a separate thread, providing a substantial performance improvement.

Not all browsers support multithreading, resulting in a slower experience with Google Earth. Neither Firefox nor Opera supports multithreading, despite Opera being Chromium-based just like Chrome and Edge. This is because a feature called SharedArrayBuffer was pulled from all browsers due to the revealing of security vulnerabilities. The SharedArray-Buffer makes multithreading possible; thus, Chrome implemented Site Isolation, limiting each rendering process to single-site documents. Thanks to Site Isolation, Chrome could open up the ability to use SharedArrayBuffer again, making multithreading possible.

Safari also has a solid Wasm support, but it was not until recently that Google Earth was able to be run on it. The problem with Safari was that they had not implemented support for WebGL2 [23], but as of October 2020, they have enabled it by default [24], meaning Google Earth can now run there as well. The WebGL2, which stands for Web Graphics Library, is essential for Google Earth as it is a JS API that can render both 3D and 2D high-performance interactive graphics.

4.1.2 BFT in browser

BFT protocols are not inherently designed to run in browsers. To our knowledge, a completely web-based client-server implementation of a BFT protocol does not exist. However, BFT clients have been deployed to browsers to interact with a BFT service [25] and at least two BFT middleware for Web Service applications have been introduced [26, 27].

In [25] they look at the possibility of combining a web client with a BFT SMR protocol. They wanted to bring the reliability and resilience commonly found in native applications to the Web as more and more native applications transform into web applications. They address four main research questions, which are focused on the interface between the BFT SMR protocol and the web client, bootstrapping and authentication, web services' execution model and the performance of BFT web services. Their server-side implementation is based on the BFT-SMaRt protocol [28], while, due to restrictions of the browser, their client-side implementation differs from it. The WebBFT implementation is compared to BFT-SMaRT Java clients, where they send requests and responses of varying sizes. Based on the experiments performed on both WebBFT and BFT-SMaRT Java clients, the authors conclude that their implementation, in terms of latency and throughput, achieves comparable fast performance.

As mentioned, both [27] and [26] describe BFT middleware for Web Service applications. Both of them focus on using standard Web Service technology and explain the challenges of combining BFT with Web Services. The BFT-WS [27] is implemented as a pluggable module and is designed to operate on top of SOAP, which is the standard messaging framework. Their experiments confirm that BFT-WS only contributes a moderate amount of overhead due to the mechanisms' complexity. The Thema [26] middleware provides a new, structured way of building Web Services that are not only BFT and survivable but also fit the application model of standard Web Services. The authors perform some experiments, where they compare Thema to a gSOAP [29] version, which is non-replicated and non-BFT. From their performance analysis, the authors conclude that the results of Thema and gSOAP are comparable.

4.2 Analysis

In this section, we perform an analysis of Wasm and relab/hotstuff to see what features are required and how we handle the obstacles described in Section 3.6. We have also included an analysis of possible execution environments.

4.2.1 Requirements

The relab/hotstuff is built with Go and designed to run on server hardware. It is making use of many of the tools available on a standard hardware platform. Configuration files hold all replica configurations, and both private and public keys for each replica were stored in separate .key files. The gRPC package handles the networking. This package creates the listeners and opens connections between each participating replica. TCP connections are opened and are configured with HTTP/2 to create streams. Each replica act as both server and client, sending and receiving RPCs.

Each replica has a set of public and private keys to sign and verify certificates during operation. Cryptographic calculation support is required for HotStuff to function correctly.

In essence the requirements for relab/hotstuff are the following.

- Accessible configurations
- Network connectivity
- Cryptographic functions

As we are designing WasmStuff, we need to ensure that these requirements are met in one way or another. When compiling to Wasm the ability to use file systems are lost since this is not supported. We handle this by integrating the required configurations and keys into the code itself. A secondary reason for this way of handling it is to allow for easier demonstration and testing of the system. In Chapter 8 we discuss more secure and versatile alternatives.

For a replica to verify certificates from other replicas and create certificates themselves, they need to perform cryptographic functions. Not all Wasm compilers can successfully compile these functions to Wasm. The compiler TinyGo [9] does not support these, and we have therefore not been able to use it. Fortunately, the official Go compiler for Wasm does support cryptography.

4.2.2 Networking with WebAssembly

To get the HotStuff protocol running, we need a method to exchange messages between the leader and the other replicas. The Wasm sandbox environment that runs in the browser lacks support for accepting network connections. The initial attempt to run the Wasm code without any changes proved unsuccessful. The execution relies heavily on network listeners to accept incoming network connections. Due to the inability to accept connections directly, we examined alternative solutions.

The relab/hotstuff implementation was created in Go and was using gRPC and Gorums, which is a wrapper for gRPC. RPCs are a technique that is used primarily for client-to-server communication and allows for the execution of commands on a remote server. A browser lacks the required features to allow it to act as a server, and the same goes for the Wasm environment in the browser. By implementing a WebSocket package in Go, connectivity could be established from within the Wasm instance [30]. However, this cannot work around the lack of socket acceptance functionality. The Wasm code can act as a client and can only dial out but not receive connections. In order to avoid performing extensive adjustments, a possible workaround for this is to implement a proxy server. This proxy server would answer connections from the Wasm instances and connect the replicas, which would result in established end-to-end connectivity.

A modification of the gRPC package was performed to replace the listeners with WebSocket dialers. This modification was in part successful as gRPC managed to reach the proxy server. Nevertheless, when trying to connect two ends, gRPC was not able to open an HTTP/2 tunnel. Even if this attempt had been successful, the resulting system would be very vulnerable due to a single point of failure in the proxy server. Therefore, due to both it being unsuccessful and the vulnerabilities, we abandoned the idea of gRPC proxying.

The WebRTC project is an experimental project that allows for peer-to-peer communication between browsers. The WebRTC protocol is still restricted by the fact that browsers cannot listen on a socket. To establish a WebRTC communications channel, the initiator starts by connecting to a Session Traversal Utilities for NAT (STUN) server. The N in STUN is an abbreviation for Network Address Translation (NAT). The Interactive Connectivity Establishment (ICE) protocol use the STUN server to discover the Internet Protocol (IP) address and port that the NAT protocol has allocated. After gathering the address and port of a peer, the ICE protocol creates a Session Description Protocol (SDP) offer and shares it with the other peer.

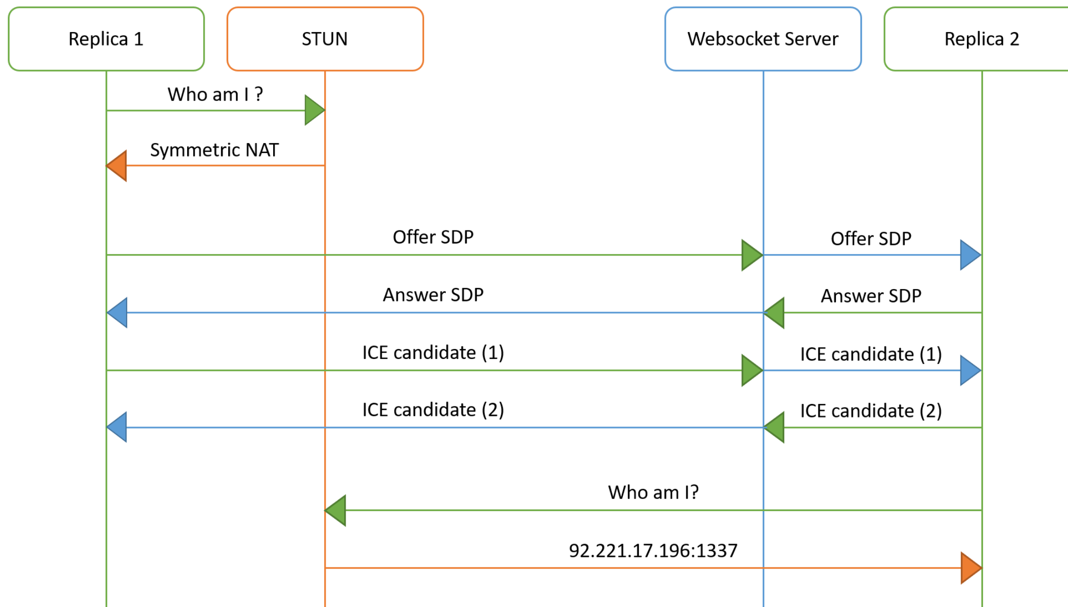


Figure 4.1: Illustration of how WebRTC connectivity is achieved.

On the server hosting the webpage, WebSockets (WSs) are implemented to allow the peers to exchange SDP messages. Figure 4.1 illustrates how the WebRTC process progresses. When the remote peer has received the initial offer and connected to the STUN server, they generate an answer. This answer is exchanged with the initiating peer over a WS connection via the webserver. Both peers take the offer and answer and configure them as local and remote session descriptions. The channel is now ready. With the channel established between the peers, all traffic moves directly from one peer to the other. There is no intermediary server between them [31].

4.2.3 Execution environment

As of today, there are two possible methods for executing a Wasm binary, one being in browsers and the other is through the use of Wasm runtimes.

Since the execution of Wasm in the browser is the primary goal and was implemented first, this has more advanced features and more extensive testing. All the major browsers have already released support for Wasm 1.0 [7]. Using a browser as the primary execution environment for this BFT system means that the number of prerequisite installs and configurations would be approximately equal to zero. Just install or open a Wasm supporting browser and go to the IP address hosting the BFT system files. The execution of a Wasm binary is done through the us of JS and the browsers JS engine.

A browser can run Wasm binary instructions due to a new baseline compiler that the browsers have introduced specifically for Wasm [32]. Wasm is compiled to a bytecode by this new baseline compiler, which can then be understood by a JS interpreter. Previously the JS engine V8 used TurboFan as their compiler. However, as the backend of the compilation process consumed considerable time and memory a better solution was needed. V8 introduced Liftoff [33] in 2018 as the new baseline compiler. To reduce the startup time for Wasm, the bytecode generation needs to be as fast as possible. This is the goal of Liftoff. By constructing an intermediate representation and generating machine code in a single pass over the bytecode, Liftoff uses less time and memory overhead compared to TurboFan [33]. The V8 engine will pass the .wasm file through Liftoff, which creates unoptimized bytecode. Then this code is passed to TurboFan with the bytecode of the JS code, where it is optimised. Note, it is this optimised code that makes it possible for Wasm to run at near-native speed [32].

As mentioned in Section 3.6, we experienced resource starvation of the browser. The number of system calls from within the Wasm instance to the browsers UI is what caused the starvation. By removing most of the prints from the code and clearing the console log on an interval, we prevented starvation from happening. To reiterate, the prints were not the specific problem, but the general amount of system calls from within the Wasm instance to the browsers UI was what caused the resource starvation.

To be able to run Wasm outside the Web, a runtime is necessary. There are many different runtimes to choose from, but as we are using Go we looked into using Wasmtime [34]. Furthermore, Wasmtime is one of the few runtimes that have support for Go. Wasmtime is one of the fastest standalone runtimes for Wasm. In addition to a runtime WASI is needed to run Wasm outside the Web, as mentioned in Section 3.4. Wasmtime is an excellent choice for small programs, as it will compile and execute them incredibly fast. When we tried to use Wasmtime for WasmStuff we experienced some issues. When instantiating a new Wasm module with Wasmtime, we need to provide the imports that are found in the .wasm file [35]. Namely, Wasmtime requires all imported functions of a .wasm file to be re-defined in the main file you are going to run the .wasm file from. Due to WasmStuff importing many different packages, and thus many different functions, using Wasmtime to run WasmStuff outside the Web was not feasible.

Due to the limited time of this thesis, we decided to put our focus on browsers and left the idea of using runtimes and WASI behind.

Chapter 5

Design and Implementation

As mentioned, we are adapting and implementing an existing BFT SMR protocol in Wasm. To our knowledge, gained from analysing similar approaches to Wasm and BFT systems, we are implementing the best solution as of today.

5.1 Design overview

We propose WasmStuff, an adaptation of the relab/hotstuff protocol running in the browser. For serving the content to the Web, we have an HTTP file server created using Go.

The protocol itself is residing in a Wasm module created from the Wasm file we compile from Go. Due to the sandboxing of the Wasm instance inside the JS VM and browser restrictions, we are unable to perform networking traditionally. By making use of Pion/WebRTC, an external Go package, a complete WebRTC implementation becomes available for use in Go. When compiling to Wasm, the WebRTC package has implemented an API wrapper for the WebRTC JS API. The package makes use of the internal Go package *syscall/JS* to perform calls and execute JS from within Wasm.

The leader and each of the replicas create WebRTC datachannels between themselves. Due to the lack of connection acceptance capability in browsers, two WasmStuff replicas cannot establish the connection entirely independent. WebRTC uses SDP messages to establish connections but needs a way to exchange these. The same code running the fileserver also acts as a WS server to exchange SDP messages. With WebRTC datachannels established, WasmStuff replicas running in Wasm no longer require the fileserver or WS server to function.

5.2 Web Server

The primary purpose of the web server is to deliver the required HTML, JS and Wasm files to those who wish to partake in the BFT system. Additionally, the web server acts as an exchange service for SDP messages between peers.

The relab/hotstuff codebase that we base WasmStuff on uses Go; thus, we chose to implement the server using Go for simplicity and convenience. When the server starts, the first thing that happens is we initialise a data structure to hold SDP offers and answers. The next step is starting the HTTP fileserver to serve files from the current directory. With the Go language, it is straightforward to start the fileserver as a new thread by using the keyword *go* to execute the function as a separate goroutine. This way, the server will continuously listen for and serve file requests while the rest of the code executes. A secondary HTTP listener is responsible for serving WS connection requests.

Listing 5.1: Parameters for accepting WS connections

```
1  func (s wasmServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
2      opts := &websocket.AcceptOptions{OriginPatterns: []string{"*"},
3          Subprotocols: []string{"*"}}
4      c, err := websocket.Accept(w, r, opts)
5      if err != nil {
6          s.logf("%v", err)
7          return
8      }
9
10     ctx, cancel := context.WithTimeout(context.Background(), time.Minute)
11     defer c.Close(websocket.StatusInternalError,
12         "WebSocket has been closed")
13     defer cancel()
14     conn := websocket.NetConn(ctx, c, 1)
```

In Listing 5.1 the configuration parameters for WS handling can be seen. For simplicity and flexibility, the server accepts all origin patterns and subprotocols. Incoming WS connections are accepted and handled by the WS server. Valid incoming connections always send a command to the WS server. Command messages are valid if they include either a command or an SDP offer/answer including the ID of a WasmStuff replica. Other valid command messages include only the command to be executed and a WasmStuff replica ID. In Table 5.1 all commands available for execution over WS can be seen.

Command	Data
actpass	SDP Offer and its own replica ID
active	SDP Answer and ID of receiving replica
recvOffer	Its own replica ID
recvAnswer	Its own replica ID
removeOffer	Replica ID to remove offer for
removeAnswer	Replica ID to remove answer for
purgeDatabase	Its own replica ID

Table 5.1: Possible command phrases and included data of a WS server.

When the web server receives an incoming connection, it answers and establishes the connection before reading from the connection. The received message is a concatenated series of string representations of the command and data. The server separates the message for further processing. For each command phrase, a different set of actions are performed. *actpass*, *active*, *recvOffer* and *recvAnswer* are the driving commands, while *removeOffer*, *removeAnswer* and *purgeDatabase* are supporting commands. If the command phrase is either *actpass* or *active*, then it takes the message data, the SDP offer or answer, and stores it in a map with the replica ID from the message as the key.

The commands *recvOffer* and *recvAnswer* are request commands. The browser can only initiate the WS connection since they cannot answer incoming connections. The solution is the *recvOffer* and *recvAnswer*, where the Wasm instance in the browser initiates the connection and tells the WS server that it is ready to receive waiting offers or answers. By not having constantly open WS connections and blocking threads waiting on incoming messages, the system is more optimised for the Web. Upon processing these commands, the WS server finds the offer or answer stored on the provided replica ID and delivers it to the requesting replica. The remaining commands are clean up commands used to remove old or already used SDP messages from the web server. Removing old SDP messages avoids the creation of malformed or unwanted connections.

5.3 Website

The primary platform for WasmStuff is the browser, and this provides an execution environment for the Wasm module. For demonstration purposes, we preconfigured the system with an easy setup of a four replica system. A set of buttons starts and configures each browser window so that they are ready to run as WasmStuff replicas. After choosing which replica to start, the user has two options: to run a benchmark of WasmStuff or to play chess against one of the other replicas. All replicas have to choose the same option to ensure proper functionality.

When choosing the benchmark option, the user can specify how many commands they want to execute by entering a number in the input field *Number of Cmds*. An empty input will default to 1000 commands. The button titled *GO* will execute the .wasm file, and WasmStuff will start. After a stabilisation time, to let all replicas connect, the message traffic is started to activate the BFT system fully. The user has an option of creating a command by typing in the *Command* input field, which will be sent to the leader replica and handled accordingly. The reason for including the possibility to send commands is to simulate incoming commands from clients.

When choosing the chess game option, there is no need for further input to start the .wasm file and initiate WasmStuff. When the replicas have completed their connection phase, a function within Wasm activates three buttons in each browser window. The buttons allow the user to pick which replica to challenge to a game of chess. When a replica chooses an opponent, they send a message to the opponent instructing the replica to start a chess game in the browser window. The two remaining replicas get instructed to draw a spectator board displaying all the moves performed by the players. When a player makes a valid chess move, the move is sent to the WasmStuff protocol in the same matter as the user commands in the benchmark option. Once the protocol executes the command, the four replicas will update their displayed boards with the new move.

We have included the possibility of purging the WebRTC SDP database and downloading an MD5 checksum of the .wasm file in the top left corner. An in-depth user guide, with images, can be found in Appendix A.

5.4 WebRTC for WebAssembly

To enable communication from within the Wasm instance, we are using the Pion WebRTC package for Go [36]. This package provides the ability to keep all the necessary connection establishment processes contained within the Wasm instance. By moving the networking into Wasm, we gain a higher level of control over the process as well as cleaner and more readable code. Moving the networking provides quite the improvement over our first attempt, where we exchanged messages with JS in the browser. This WebRTC package is a complete and pure Go implementation of the WebRTC protocol.

When compiling for conventional platforms like Windows, Linux and macOS, the WebRTC package has implemented the entire protocol using pure Go. However, due to the lack of support for socket listening, Wasm is currently restricting complete networking control and support from within a Wasm instance. Therefore, when compiling to Wasm this package is only a wrapper for the WebRTC JS API. There is no need for any changes to the logic when choosing different compilation targets. The only changes needed is the removal of JS system calls and adding command-line input of options. With those small surface changes, the system compiles to non Wasm platforms. These changes allow us to execute and run WasmStuff conventionally without Wasm outside the browser using the same code. While this is outside of our intended target of diversification through Wasm, it still adds to the overall goal of diversification of BFT systems.

When the Wasm instance is loaded, the first thing that happens is the replicas fetch their identifiers from the webpage through JS system calls. After this, the replicas are ready to establish connectivity. WasmStuff uses all-to-all connectivity to allow for rapid leader rotation. The WebRTC datachannel creation starts with one replica generating an SDP offer for the other replica to receive and generate a corresponding SDP answer to send back. Specifying an order for the replicas to follow ensures that they follow the protocol connection process. Replica 1 will act as the listener and will request offers from the WS server, while the other replicas will act as diallers and generate their offers and send them to the WS server.

The replicas perform the following procedure for each peer-to-peer connection they establish. They begin by configuring a list of STUN servers. To create the underlying peerconnection object, they have to use the configured servers from this list. The dialling replica will then create a datachannel for the peerconnection, while the listening replica will register an event handler for incoming datachannels.

The dialler will contact the STUN server to create an SDP offer for a new connection. This offer gets configured on the replica as the local connectivity description before sending it to the WS server with the *actpass* command and its ID, as seen in Listing 5.2. The function DeliverOffer takes the offer as input and prepares the full commandstring with ID before sending.

Listing 5.2: Offer creation and delivery

```
offer, err := peerConnection.CreateOffer(nil)
    if err != nil {
        panic(err)
    }

    // Create channel that is blocked until ICE Gathering is complete
    gatherComplete := webrtc.GatheringCompletePromise(peerConnection)

    err = peerConnection.SetLocalDescription(offer)
    if err != nil {
        panic(err)
    }

    <-gatherComplete

    DeliverOffer(peerConnection.LocalDescription().SDP)
```

At the same time, the listening replica will connect to the WS server with the *recvOffer* command. After receiving the offer, the replica will configure the offer as the remote connectivity description. Following this, the replica will contact the STUN server to create an answer to complete the connection. The replica configures their local connectivity description to be the created SDP answer. The answer generation is not an instant process and has a gathering period where connection candidates are collected. When the gathering phase has completed the candidate retrieval, it will then send the complete SDP answer to the WS server with the *active* command and the ID of the dialling replica.

In the meantime, the dialling replica has been requesting the answer from the WS server using the *recvAnswer* command. Upon receiving the answer, the replica configures this as the remote connectivity description completing the connection process. After configuring the remote connectivity description, the peerconnection establishes, and the datachannel is ready for use. Both replicas are now fully connected and can exchange messages with each other. For both replicas, we have added an event handler that triggers when a message is received. Pion WebRTC features two ways to send and receive messages, byte slices or strings. In Go, a slice is an object type allowing dynamic usage of arrays, automatically creating larger arrays if needed.

WasmStuff protocol messages are exchanged in byte slice format, while the networking support commands we chose to exchange in string format for readability. The WebRTC package performs the conversion to byte slice in the background for string messages as well, but adds a flag indicating that the software can parse the message directly as a string. The support commands execute specific functions to complete the networking setup and do not affect the consensus protocol. When a byte slice message is received, it gets added to a message slice ready for processing by the WasmStuff protocol code.

When the first listening replica (ID = 1) has established connections with the other replicas, it sends a message to the replica with an ID one higher than itself (ID = 2). The replica sends the command *StartConnectionLeader* to the next replica. Upon receiving this command, the replica will start a goroutine, making the replica act as a listener.

The other remaining replicas will continue creating offers and sending them to the WS server, then wait for answers. This replica will act as a listener until it has established connections with all other replicas before passing the listener role to the next replica. All replicas will have peer-to-peer connections with datachannels for each of their peers when this process has finished. The last listening replica will send the command StartWasmStuff to all the others to tell them to start the protocol.

5.5 WasmStuff

This section describes the WasmStuff runtime, and we go more in-depth on the changes made to the relab/hotstuff implementation. The runtime is made for this thesis and to run the WasmStuff consensus protocol, but with minor adjustments, any BFT protocol could potentially replace WasmStuff. The versatility of the runtime is elaborated on later in this section as well.

5.5.1 WasmStuff Interface

The relab/hotstuff implementation of HotStuff made a *hotstuff* package for the backend, with types and interfaces our solution needs to implement to operate. Due to some of the changes we needed to do to the relab/hotstuff implementation, we also made some changes to the *hotstuff* package. We named the modified package *hotstuffwasm*. It is in the *hotstuffwasm* package that we find the interfaces for the consensus protocol, the pacemaker, configuration of replicas, the blockchain, and more, which we implemented in our solution.

In Table 5.2 some of the methods implemented by the consensus interface are described.

Method	Description
LastVote	Returns the view in which the replica last voted
HighQC	Returns the highest QC known to this replica
Leaf	Returns the last block that was added to the chain, which is also the block with the highest view known to the replica
Blockchain	Returns the blockchain with all the blocks known to the replica
CreateDummy	Creates a dummy-block at View+1, which is used when the replica times out
Propose	Starts a new proposal
NewView	Creates a NewView message
OnPropose	Handles incoming proposals, including the leaders own proposal
OnVote	Handles incoming votes
OnNewView	Handles incoming NewView messages
Synchronizer	Returns the replicas pacemaker

Table 5.2: Methods in the consensus interface

As seen in Table 5.2 the consensus interface has a method for getting a blockchain. The blockchain is used to store all proposals, made by a replica, as blocks.

Method	Description
OnPropose	Restarts the timer of the pacemaker
OnFinishQC	When a new QC has been, the pacemaker beats
OnNewView	When a valid NewView message has been received, the pacemaker beats
Init	Initialises the pacemaker with the consensus protocol
Start	Sets the timer of the pacemaker, and starts a proposal if the replica is the leader
Stop	Stops the pacemaker and the timer

Table 5.3: Methods in the synchroniser interface

The pacemaker (or synchroniser) is what keeps WasmStuff running and alive. It ensures that all replicas are on the same view and is responsible for starting new proposals on the leader replica. The pacemaker does so by beating, here meaning it will start a new proposal. Whenever a replica makes a new proposal, it notifies the pacemaker, which resets the timeout timer. In Table 5.3, descriptions of the pacemakers methods are included.

The first thing to happen when starting WasmStuff is the initialisation of a replica. The parameters of a replica (server) are displayed in Table 5.4.

Variable	**Description**
ID	ID of replica
Addr	Address of replica
Hs	HotStuff consensus protocol
Pm	Pacemaker
Cfg	Configuration of replica
PubKey	Public Key of replica
Cert	Certificate of replica
CertPEM	Certificate PEM of replica
PrivKey	Private key of replica
Cmds	Command buffer where incoming commands are stored
RecvBytes	Slice of bytes of incoming messages

Table 5.4: Struct of replica server

The configuration, *Cfg*, of the replica is instantiated with a ReplicaConfig data structure, holding information about all the other replicas as well as the configuration of the current replica.

5.5.2 Runtime

For this thesis, we run the WasmStuff protocol with four replicas. Four is the minimum number of replicas needed to ensure that the BFT requirements of $N > 3f + 1$ can be met, and it is sufficient for testing purposes. The number of replicas is hardcoded into the setup of networking for the replicas. However, the number of replicas does not matter for handling the different messages sent between them. It is possible to create a dynamic setup of this, which we describe in Chapter 8.

When starting a replica, it will sleep until an ID is available with which to configure itself. Once the replica has set its ID, the pacemaker is created and instantiated with a leader rotation and a fixed timeout value. With the pacemaker and the other crucial variables instantiated, the initialisation of the replica is done.

Then, we set up the connections between the replicas, as described in Section 5.4, and start the pacemaker. When the pacemaker starts, it checks if the replica is the leader, and if so, it will start a proposal. The proposal returns a byte slice representation of a proposal message string. A channel receives this byte slice, which will trigger a select case on the leader. At the end of the start function, the pacemaker will start a NewViewTimeout in a separate goroutine, which will trigger a NewView if the timer runs out.

Now that we have everything set up and it is ready to use, the runtime checks if the replica is the leader or a normal replica. There is an endless loop with a select statement for both the leader and the normal replicas to determine the next action to take. In Figure 5.1, an overview of the leader replica's runtime is illustrated.

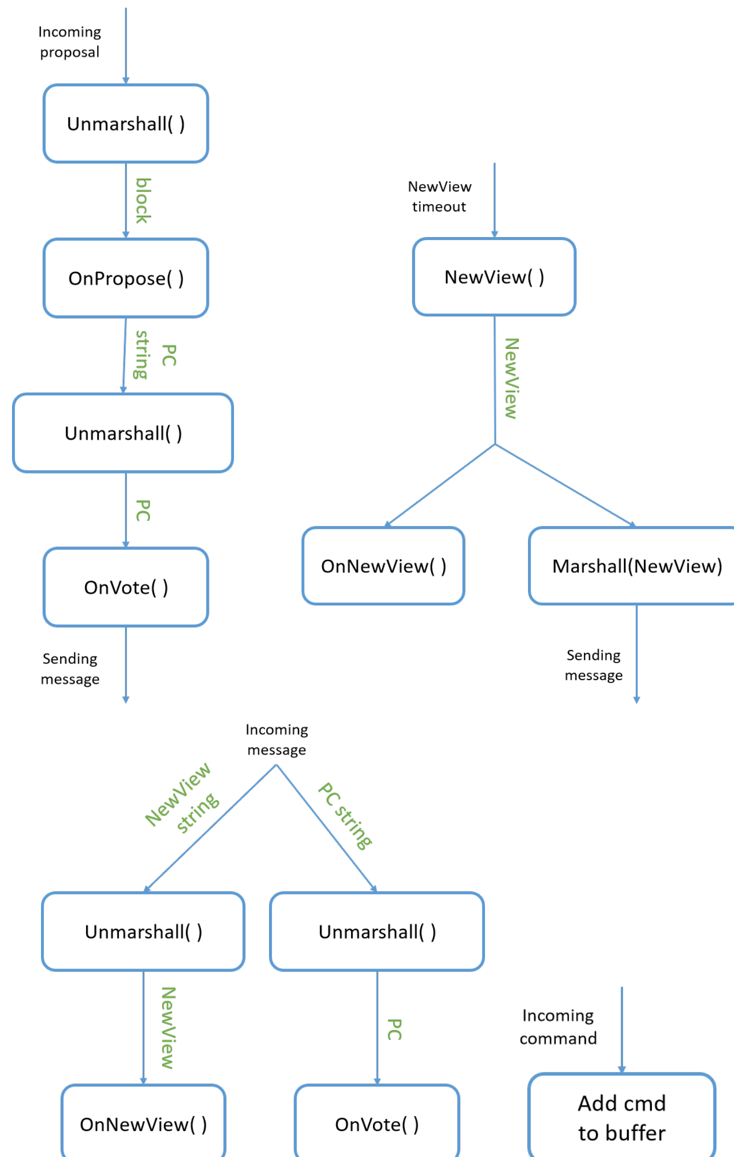


Figure 5.1: Overview leader replica's runtime

The four different flowcharts in Figure 5.1 illustrate the four different cases in the select statement for the leader. As mentioned earlier, the pacemaker will send the proposal, in the byte slice representation of a string, to a channel to indicate a waiting proposal. The leader then reads this channel, and as seen in the top left of Figure 5.1 the leader unmarshalls the string and performs the necessary operations to it before sending the received proposal to the other replicas for them to sign.

The top-right flowchart shows the leaders response to a NewView timeout. Whenever the leader receives a message from one of the other replicas, the message could be one of two things: a NewView or a Partial Certificate (PC). The leader handles the two different messages as follows:

NewView: the incoming message is unmarshalled to a NewView and the OnNewView() function is called, where the highest Quorum Certificate (QC) is updated. If a quorum of NewView messages has been received the leader calls the NewView() function to start a new View.

PC: the incoming message is unmarshalled to a PC and the OnVote function is called. If a quorum of PCs is received, a QC is created and the Pacemaker is notified to beat again to start a new proposal.

If the leader receives a client Command, it will store the command in a command buffer. The next time this replica is the leader, it will propose this command.

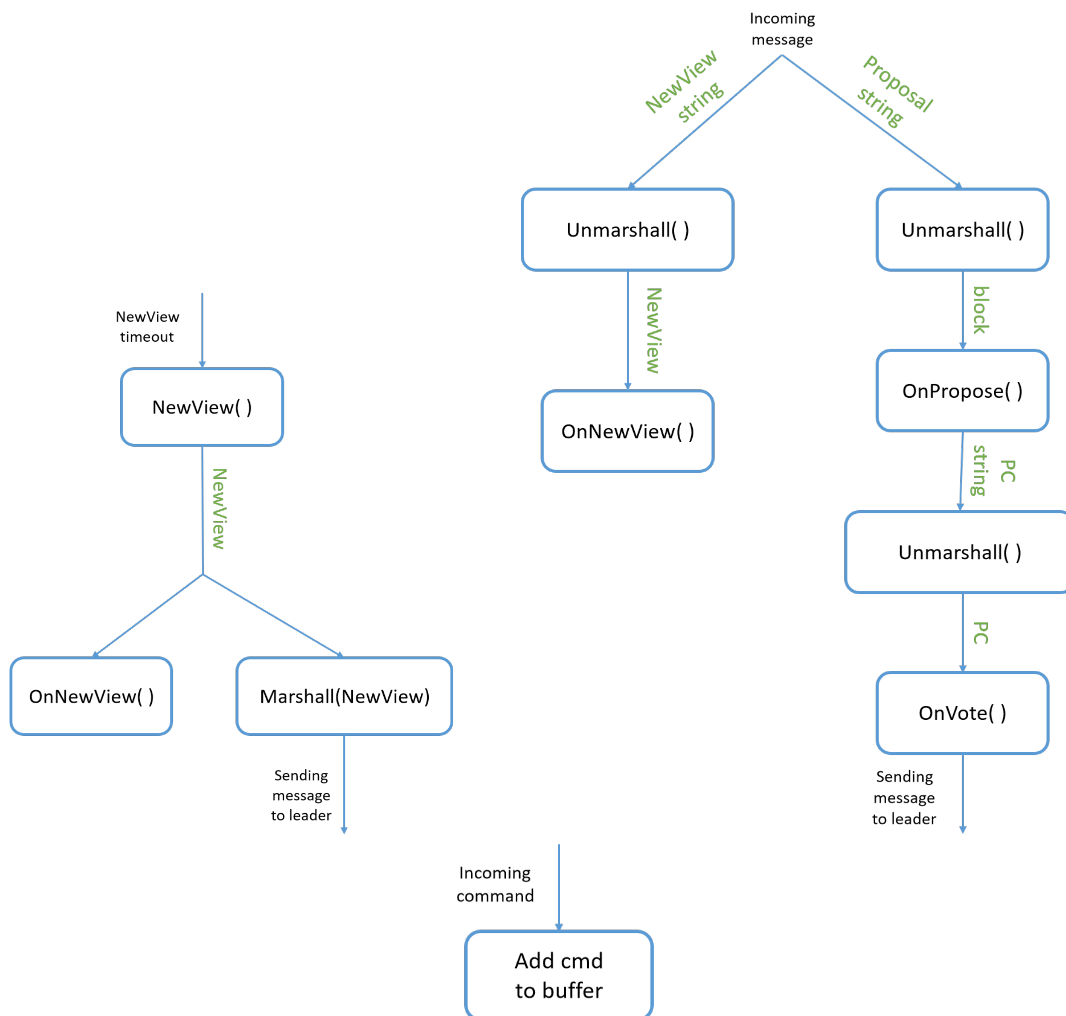


Figure 5.2: Overview of normal replica's runtime

As illustrated by the flowcharts in Figures 5.1 and 5.2, the runtimes for both the leader and the normal replicas are relatively similar. A normal replica that receives a client COMMAND will also store the command in a buffer. As with the leader, the next time this replica is the leader, it will propose this command. If there is more than one command in the command buffer, the next leader will propose the command according to First In, First Out (FIFO).

A normal replica also handles two different messages. The leader sends either a PROPOSAL for the normal replica to handle or a NEWVIEW message to indicate a new VIEW. A normal replica will handle these two messages as follows:

NEWVIEW: The replica will update its highest QC and create a NewView message, which will then be sent back to the leader.

PROPOSAL: The incoming message is unmarshalled into a BLOCK, which is sent as a parameter to the ONPROPOSE() function. The returning signed PC string is then sent to the leader.

As mentioned at the beginning of this section, this runtime is specific for WasmStuff. That said, with minor adjustments, a similar BFT protocol could replace WasmStuff. The networking part, which would be the most useful part of the runtime, only needs to be defined with a different IP address for the WS server. The leader and the normal replicas runtimes would need modifications regarding what messages to receive and what to do with them.

Listing 5.3 includes a part of the runtime for the Leader replica. Here one can see how we implement these flowcharts in our solution. As seen in Listing 5.3, upon receiving a message on the *received* channel, the leader checks the incoming message, *recvBytes[0]*, to see what kind of message it is. The received message is a byte slice representation of a string, which we convert back to a string. The first part of the message includes the type of message, making it easy to differentiate them. After recognising the type of the message, the replica will handle the message accordingly.

Listing 5.3: Behaviour of Leader replica upon receiving a message

```
case <-recieved:
    recvLock.Lock()
    newView := strings.Split(string(recvBytes[0]), ":")
    recvLock.Unlock()
    if newView[0] == "NewView" {
        recvLock.Lock()
        msg := StringToNewView(string(recvBytes[0]))
        recvLock.Unlock()
        srv.Hs.OnNewView(msg)
        recvLock.Lock()
        if len(recvBytes) > 1 {
            recvBytes = recvBytes[1:]
        } else {
            recvBytes = make([][]byte, 0)
        }
        recvLock.Unlock()
        continue
    }
    recvLock.Lock()
    pc := StringToPartialCert(string(recvBytes[0]))
    if len(recvBytes) > 1 {
        recvBytes = recvBytes[1:]
    } else {
        recvBytes = make([][]byte, 0)
    }
    recvLock.Unlock()
    srv.Hs.OnVote(pc)
```

As seen on line 20 in Listing 5.3, if the message type was not a NEWVIEW, the string message is interpreted as a PARTIAL CERTIFICATE. The replica will then unmarshall the string to a PC, by using the STRINGTOPARTIALCERT() function. Listing 5.4 illustrates the unmarshalling of a string to a PC. We handle the marshalling manually in this thesis to be able to make the messages easily readable from JS. This manual marshalling was something we needed in the early stages of development, as we needed to be able to read the messages in JS to see what message it was to determine which replica(s) should receive it. Using a predefined package for marshalling/unmarshalling would most likely be faster and easier to implement. In contrast, manual marshalling gives us more control to quickly validate that the messages are correct. However, knowing the type of message is no longer necessary within the JS script, and thus we could have chosen to use a package for marshalling. We decided not to prioritise this as our own marshalling methods are already implemented and should not impact the performance.

Listing 5.4: Unmarshaller for a string to a Partial Certificate

```
func StringToPartialCert(s string) hotstuff.PartialCert {
	strByte := strings.Split(s, ":")
	signString := strings.Split(strByte[0], "-")

	rInt := new(big.Int)
	rInt.SetString(signString[0], 0)
	sInt := new(big.Int)
	sInt.SetString(signString[1], 0)
	signer, _ := strconv.ParseUint(signString[2], 10, 32)
	sign := *hsecdsa.NewSignature(rInt, sInt, hotstuff.ID(signer))

	hash, _ := hex.DecodeString(strByte[1])
	var h [32]byte
	copy(h[:], hash)
	hash2 := hotstuff.Hash(h)
	var pc hotstuff.PartialCert = hsecdsa.NewPartialCert(&sign, hash2)

	return pc
}
```

When unmarshalling the string to a PC, the function first splits the string into two parts. These two parts give us a string of the signature and the signed block's hash. Then we split the signature string into three parts. Here we find the two integers needed for the signature and the ID of the signer. The NEWSIGNATURE() function restores the signature, with the integeres and ID as parameters. Finally a PC is created by sending the signature and hash to the NEWPARTIALCERT() function. Similar methods are used for unmarshalling the different types of messages. See Appendix B for all marshalling and unmarshalling methods.

5.5.3 Consensus protocol

As mentioned in Section 2.2, the relab/hotstuff implementation uses Gorums with gRPC. Due to the limitations of Wasm, it is not possible for us to use gRPC in WasmStuff, and therefore WebRTC is used as its replacement for networking. A consequence of this is that the entire Gorums package has to be removed due to it relying on gRPC. When removing Gorums from the relab/hotstuff protocol, we must rewrite parts of the code.

In almost every function call within the consensus protocol, messages are sent from one replica to another, and this is where the original relab/hotstuff protocol used Gorums. Gorums was used to make quorum calls from within these functions to send messages to the other replicas. However, since these calls use gRPC, we could not keep the code as it was. To ensure the protocol is still running correctly, we let the functions return with a value instead of making calls to the Gorums framework like in relab/hotstuff. When functions return with a value, the value is handled directly by the runtime. This value is either sent to the leader/other replicas or included in a call to a different function for further processing.

Some of the pacemaker functions trigger a channel connected to the replica instead of returning a value. When this channel is triggered, the replica knows it is time to handle something specific. An example of this is when the pacemaker is triggered to make a new proposal. When this happens, the pacemaker calls the consensus function PROPOSE() and sends the returning value to the *proposal* channel on the replica. When the replica receives a proposal on this channel, it handles it accordingly.

Other than the changes mentioned above, no other changes to the consensus code were necessary.

5.6 BFT SMR Controlled Chess

The simple demonstration that prints executed numbered commands is not too exciting or visual. To present a practical example of a use case for WasmStuff, we decided to implement a simple chess game. The BFT system has to agree on each move taken by the players before the replicas execute them. Through the implementation of the chess game, we got to work closer with Wasm's ability to interact with the Document Object Model (DOM) of our webpage.

When choosing the chess option on the webpage, the replicas go through the connection process. After the connection process has completed, a replica can invite another replica to a game of chess. Clicking on one of these buttons triggers a Wasm function that interacts with the DOM to create a chessboard and start a game. Through the use of the package *syscall/js*, the Wasm instance can manipulate the DOM. To create the chessboard two JS packages are used, chessboard.js [37] and chess.js [38]. The Wasm instance handles all the logic and configuration of the chessboard, i.e. none of the chessboard code is preloaded in the HTML file.

The invited replica has the advantage of starting and will therefore play with the white pieces. Replicas that are not playing are spectators and cannot move the pieces. The chessboard.js and chess.js are two separate packages. Chessboard.js implements a board where pieces can move wherever, while chess.js contains the chess game logic. We have to combine the two to create a chessboard where we only allow legal moves.

The players can pick up and drag their pieces when it is their turn. When a player makes a move, we check whether or not that move is valid. An invalid move is indicated by the piece quickly snapping back to its original position. A valid move will cause the piece to move back to its original position slowly. Simultaneously, the replica adds the move command to its command queue. If the replicas reach a quorum on the proposed command, all replicas will perform the chess move and update their chessboards accordingly.

Chapter 6

Experimental Evaluation

In this chapter, we describe the different experimental setups and configurations that we use for testing WasmStuff. We are also testing our implementation on different platforms and hardware to examine how diverse our available execution pool is. In addition, we are also running performance tests to measure our system's capabilities across different system configurations. We are testing the relab/hotstuff implementation as a baseline and comparing the results for both systems.

6.1 Experimental Setup

We designed our Wasm runtime and WasmStuff protocol code to function on most computer architecture as well as Wasm. However, due to the *syscall/js* package, we need separate source files to allow for compilation to Wasm and Windows/Linux/MacOS.

6.1.1 Performance testing

We have designed a test to measure the capabilities and performance of WasmStuff. Our demonstration system is instructed to exchange the highest number of messages possible at all times. We measure the time it takes to process 50 commands at a time. When 50 commands have been executed each replica stores that execution time. After storing the value, the timer is reset for the next 50 commands. Upon completion of a set amount of commands, all stored times are printed to the console. We did multiple benchmark runs with different numbers of commands to process.

For the test to yield the most equal and stable results we use identical machine for each replica. All of the browsers and Windows terminal performance testing were perform on identical machines. We did not have access to identical machines running both Windows and Linux therefore the Linux tests are performed on a different set of machines.

	Browsers and Windows:
CPU:	Intel i5-2400 3.1GHz 4 core 4 thread
RAM:	8 GB
	Linux:
CPU:	Intel Xeon e5606 2.13GHz 4 core 4 thread
RAM:	16 GB

6.2 Experimental Results

We are testing WasmStuff to see how it performs in different browsers and how it performs when using browsers together with windows. To be able to compare our results, we have tested the performance of the individual browsers, as well as running it on pure Windows and Linux. The results are illustrated using graphs. It is important to note that the time axis is fitted to the individual graphs. All data used to produce the graphs are presented in tables in Appendix C.

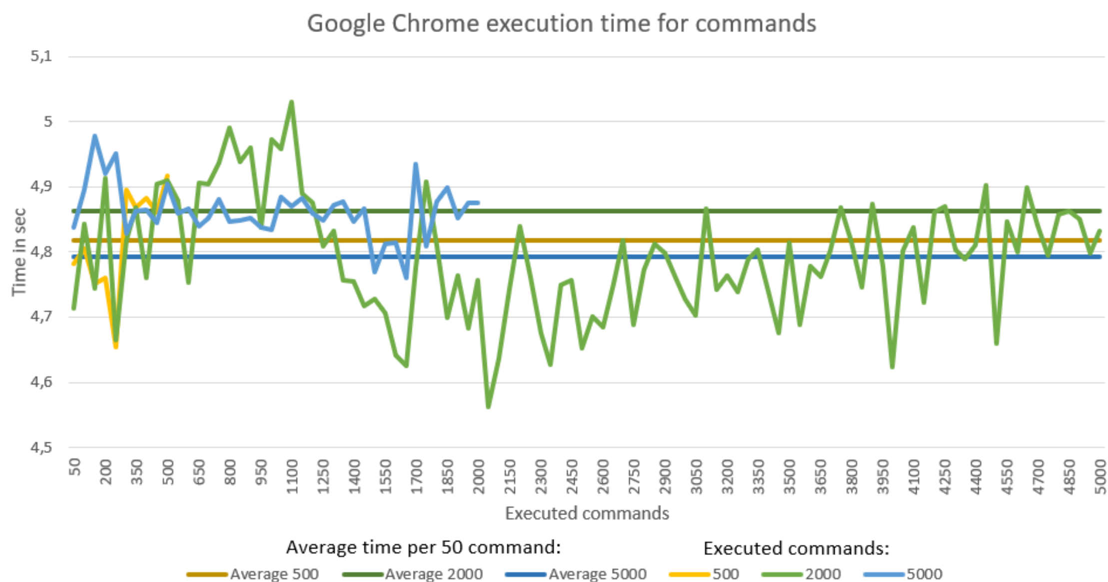


Figure 6.1: Performance of WasmStuff on four replicas in Chrome, with 500, 2000 and 5000 executed commands.

As seen in Figure 6.1, depending on the number of commands, there are some variations. Although, when looking at the averages for all three, it is clear that the number of commands does not significantly affect the results. Figure 6.1 shows that WasmStuff executes a command within a certain time Δt independently of the number of commands.

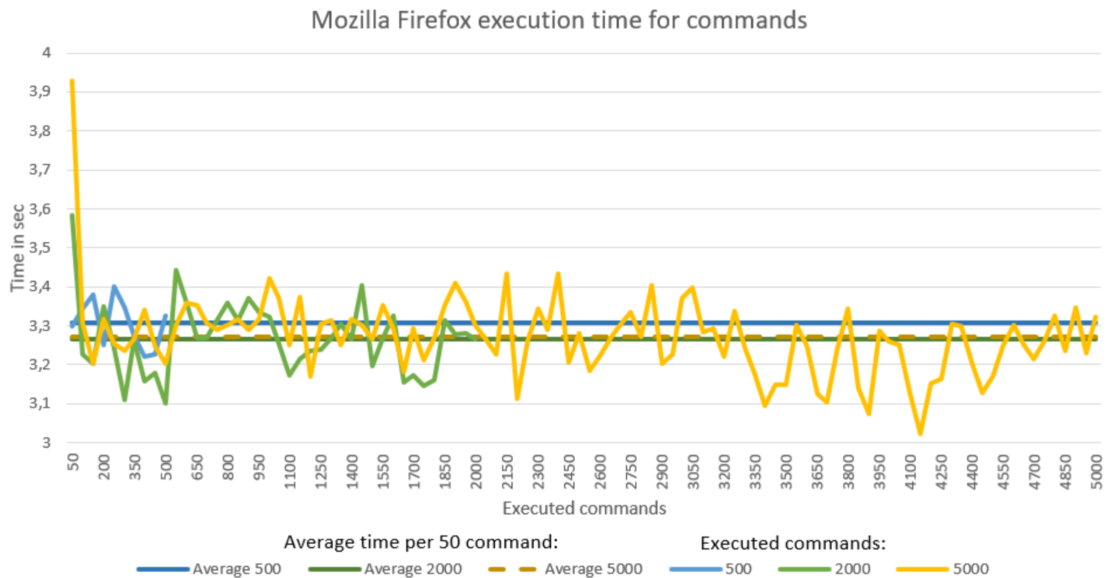


Figure 6.2: Performance of WasmStuff on four replicas in Firefox, with 500, 2000 and 5000 executed commands.

Firefox was the browser that performed the best out of the four browsers we tested. As Figure 6.2 illustrates, the average times for the three different numbers of commands centres around 3.3 seconds.

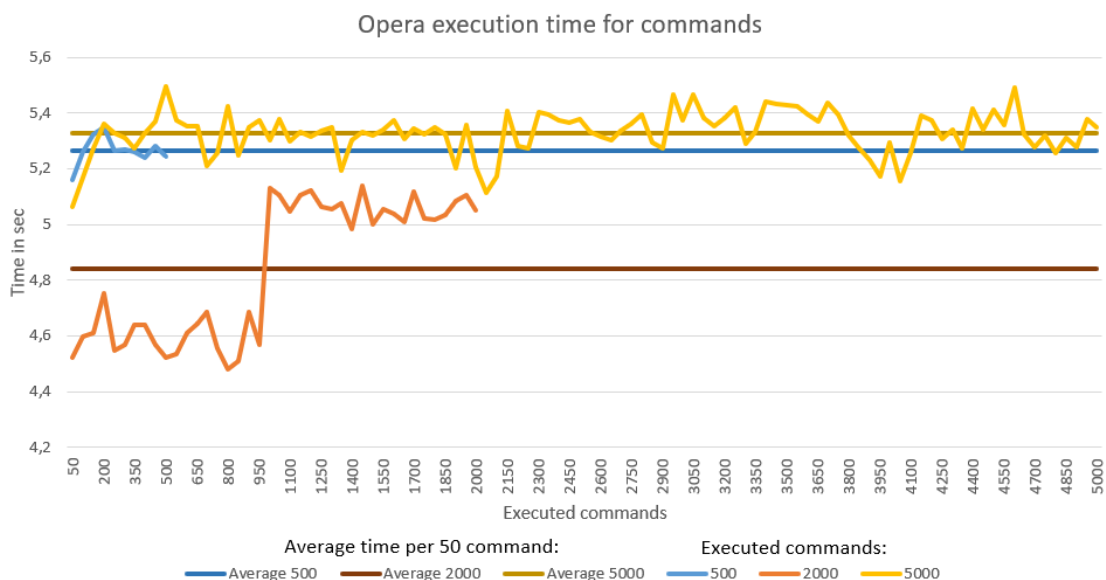


Figure 6.3: Performance of WasmStuff on four replicas in Opera, with 500, 2000 and 5000 executed commands.

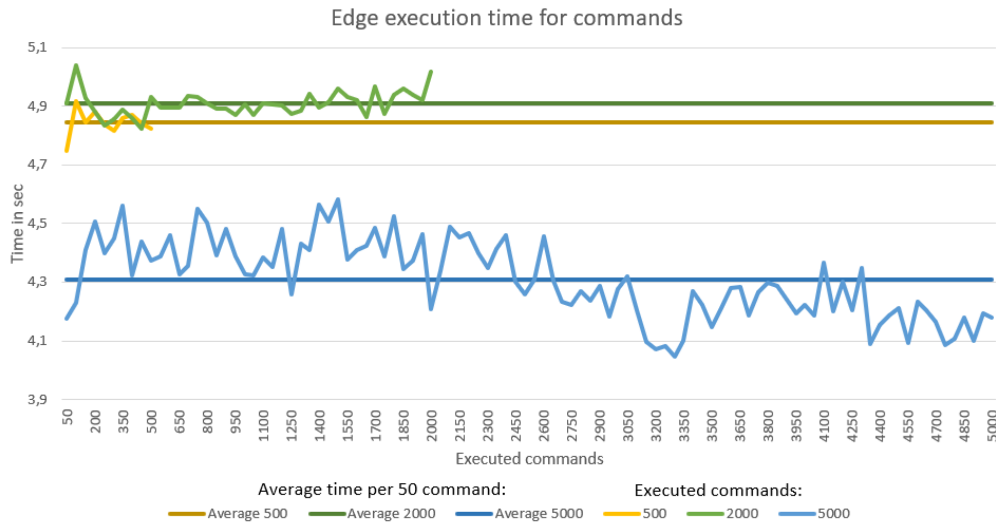


Figure 6.4: Performance of WasmStuff on four replicas in Edge, with 500, 2000 and 5000 executed commands.

In Figure 6.3 and 6.4 the execution time for Opera and Edge are illustrated respectively. Comparing these figures to Figure 6.1 and 6.2 it is clear that the averages of the different numbers of commands differs significantly. Due to unknown background processes running, we see this difference, which is out of our control.

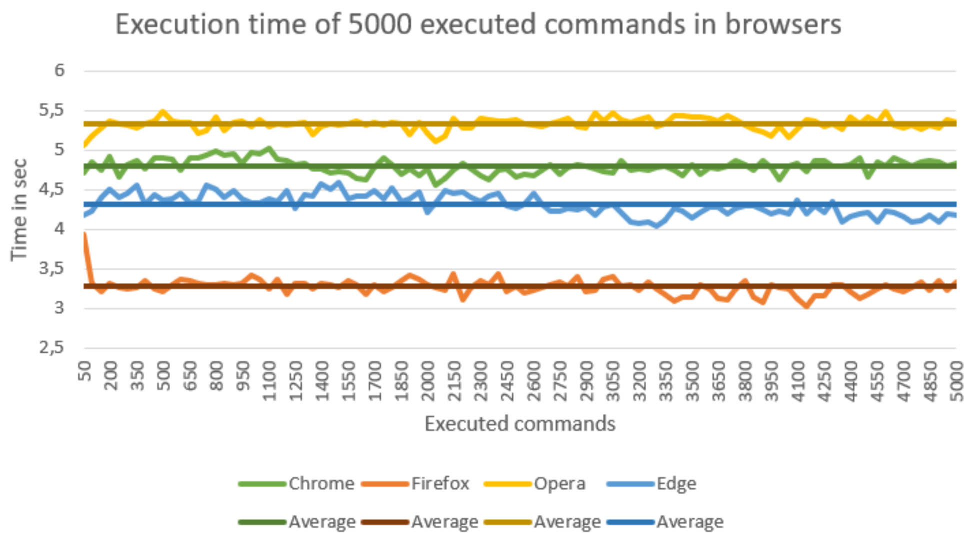


Figure 6.5: Comparison of execution times of all four browsers for 5000 executed commands.

In Figure 6.5 all testing was done with 5000 executed commands. Here we compare the different execution times for the different browsers. As mentioned before, Firefox is the browser that performed the best, beating the slowest one (Opera) by roughly 2 seconds. The other browsers differ by roughly 0.5 seconds to 1 second.

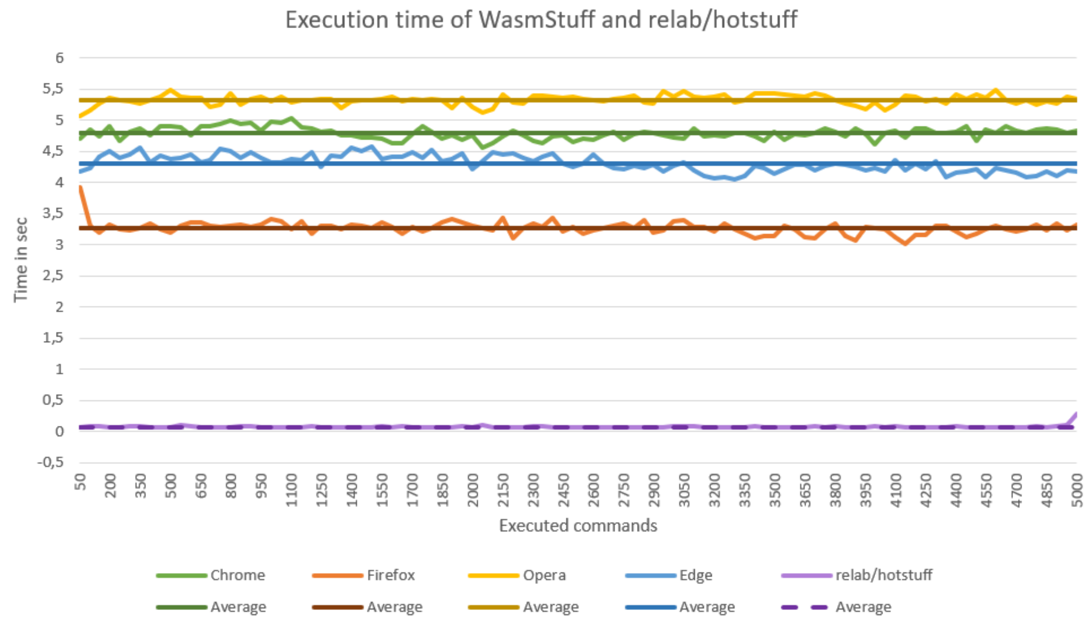


Figure 6.6: Comparison of execution times of all four browsers and relab/hotstuff for 5000 executed commands.

Figure 6.6 shows a comparison of WasmStuff run in browsers and the relab/hotstuff implementation run on Windows. It is important to note that WasmStuff was run over the Internet, and relab/hotstuff was run on localhost.

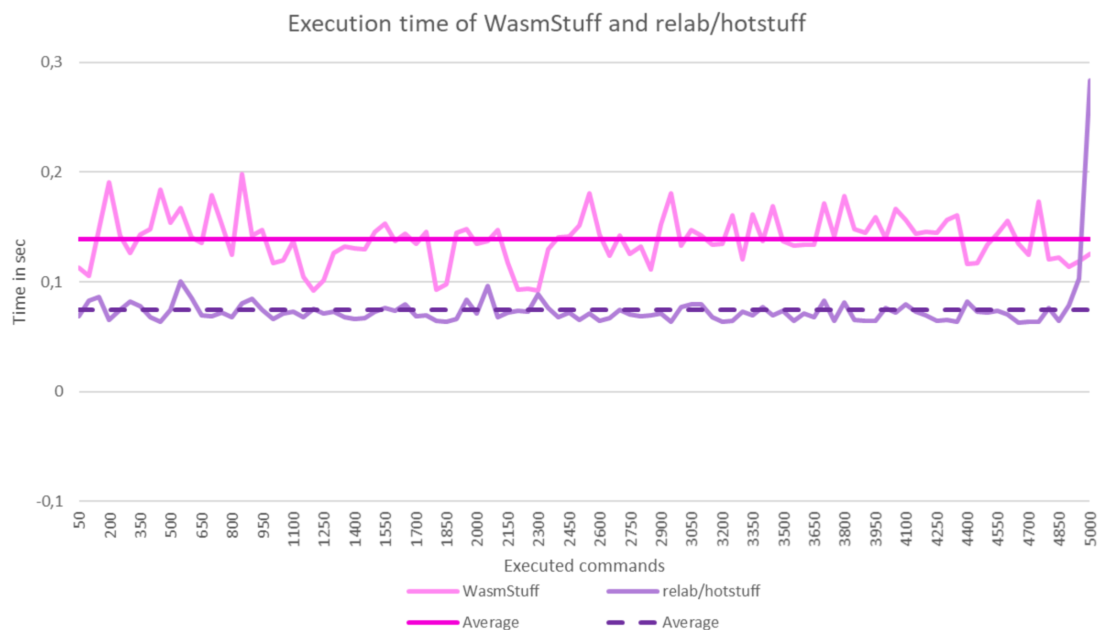


Figure 6.7: Comparison of execution times of WasmStuff and relab/hotstuff for 5000 executed commands.

To get a more accurate comparison of WasmStuff to relab/hotstuff, we ran four replicas of WasmStuff on Windows. It is clear from the graph in Figure 6.7 that our modifications to the protocol had some impact when it comes to execution time. The fact that WasmStuff was run over the Internet is the main reason for the large difference. However, considering we have diversified the protocol by bringing it to the web, the increased latency is outweighed by the achieved diversity.

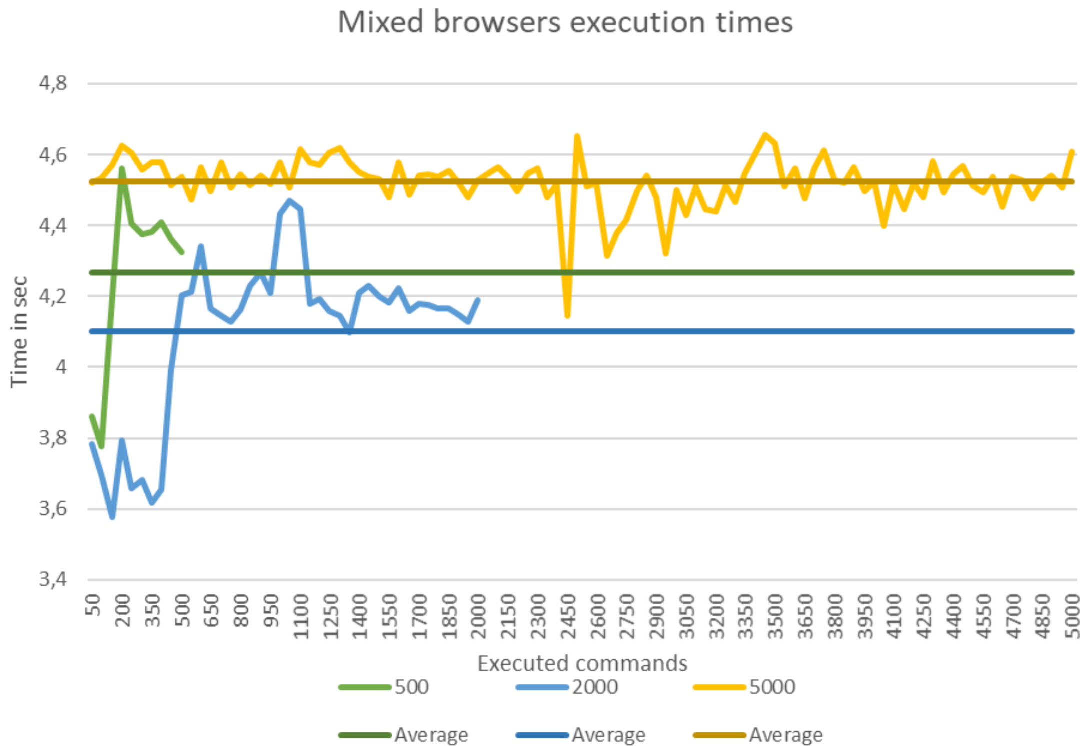


Figure 6.8: Performance of a mixed browser system with 500, 2000 and 5000 executed commands.

To further test WasmStuff, we ran tests with four different browsers running as one system, achieving even greater diversity. The browsers we used were: Chrome, Firefox, Opera and Edge. Results of the mixed browser system can be seen in Figure 6.8

Browser	Average execution time
Chrome	4,792520751 s
Firefox	3,2720525 s
Opera	5,329191183 s
Edge	4,308577754 s
Mixed	4,522804247 s

Table 6.1: Average execution time for 5000 executed commands

Table 6.1 displays the average execution times for all browsers and the system with mixed browsers. Looking at Figure 6.8 and Equation 6.1, we see that the average execution time for 5000 commands is relatively similar, as we would expect.

$$\begin{aligned} Average_{mix} &= \frac{4,792520751 + 3,2720525 + 5,329191183 + 4,308577754}{4} \\ &= \underline{\underline{4,42558575}} \end{aligned} \tag{6.1}$$

To demonstrate that our implementation works in a cross-platform system, we ran tests with different numbers of replicas running in browsers and on Windows. We tested three different combinations which are displayed in Table 6.2

	Replica 1	Replica 2	Replica 3	Replica 4
Test 1	Windows	Windows	Windows	Edge
Test 2	Windows	Windows	Edge	Edge
Test 3	Windows	Edge	Edge	Edge

Table 6.2: Test combinations for a cross-platform system

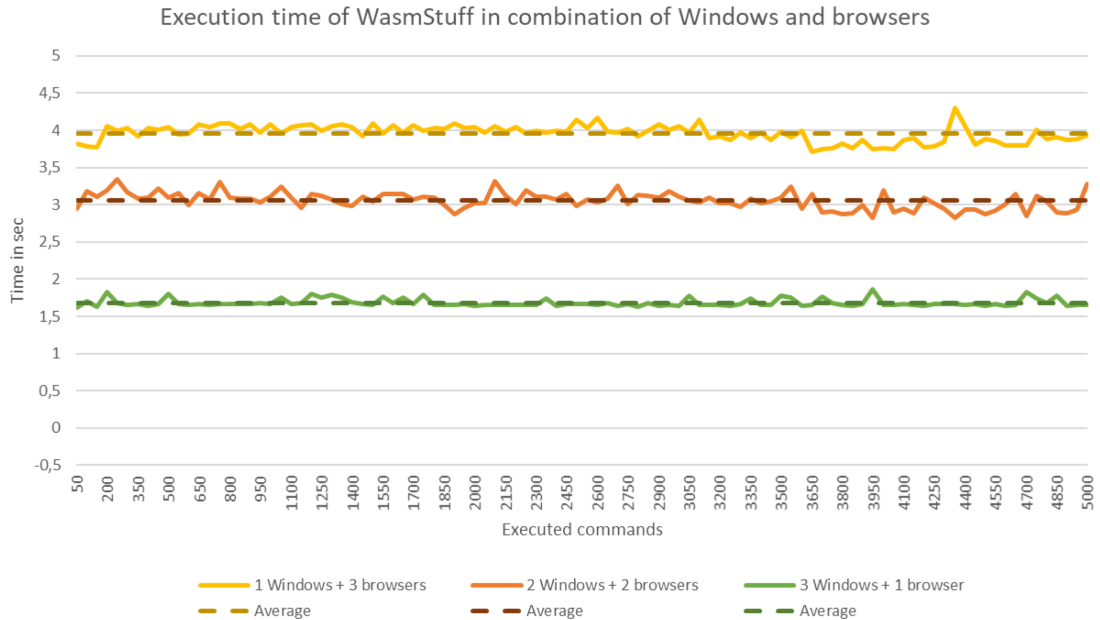


Figure 6.9: Performance of systems with different numbers of replicas running on either Windows or in a browser with 5000 executed commands.

As depicted in Figure 6.9, the more replicas running on Windows, the lower the execution time we get. This is due to replicas running in Edge will affect the execution time every time they are the leader or whenever the quorum relies on them. Remember, we need a minimum of three replies: the leader itself + 2 other replicas to reach quorum. Comparing the three test results, we can see that the fastest one is Test 1, as it has the most replicas running on Windows. In this test, Replica 4 will affect the result whenever it is the leader, but not when it is not. This only happens because Replica 1-3 can reach quorum without Replica 4.

Looking at Test 2, we see that the average execution time almost doubles compared to Test 1. The reason for this is that we now have two replicas running in Edge. These replicas will now affect the execution time whenever it is the leader and every time quorum is needed. Test 3's results have almost 1 second added latency, which comes from one more replica running in Edge and one less running in Windows.

When running WasmStuff on Windows, we experienced that the replicas were timing out frequently. As WasmStuff could run with no issues on Linux and in browsers, we assume there is an issue with the compatibility of one or more packages we use and Windows. The frequent timeouts gave us added latency, which would not have given us comparable results to relab/hotstuff. Thus, we chose to adjust the execution times with regards to the latency from timeouts.

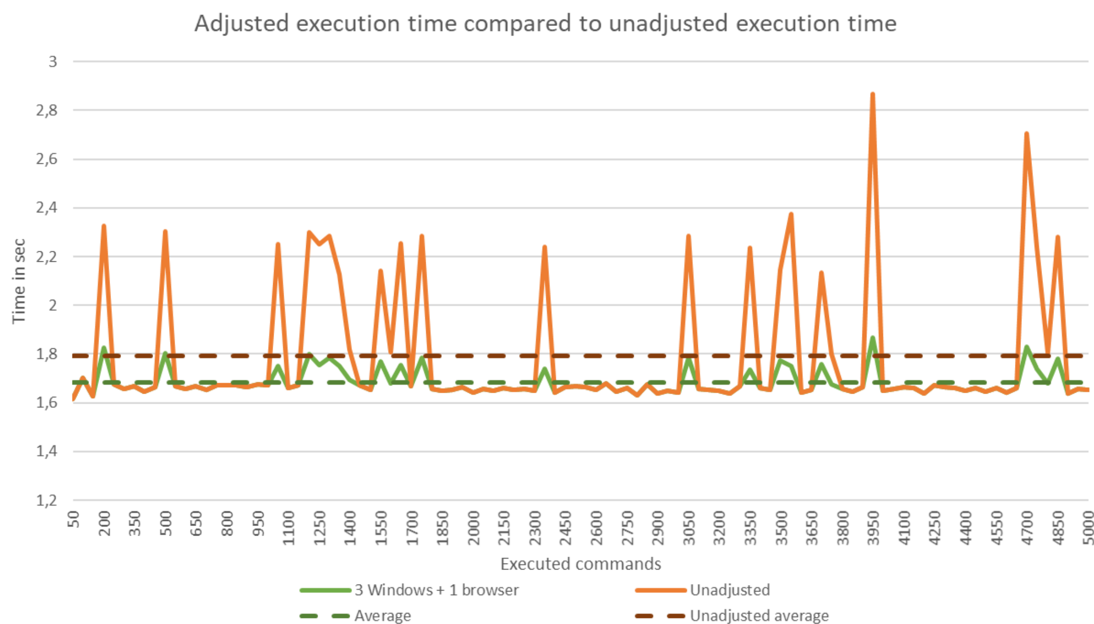


Figure 6.10: Comparison of adjusted and unadjusted execution times with regards to timeouts.

Figure 6.10 includes both adjusted and unadjusted execution times for one of the cross-platform tests. Looking at the unadjusted line, we can see how the timeouts cause spikes all throughout the graph. Even with the relatively frequent timeouts the average execution time only takes a slight hit. Despite the system timing out, it is important to note that the system is still functioning correctly.

6.2.1 Diversification analysis

Through our experiments, we witnessed WasmStuff running correctly on four major browsers. Table 6.3 displays the different browsers WasmStuff was tried to run in.

Browser	WasmStuff
Chrome	✓
Firefox	✓
Opera	✓
Edge	✓
Safari	✓

Table 6.3: WasmStuff browser compatibility

As seen from the table above, Table 6.3, WasmStuff is compatible with all major browsers. For a while, we had an issue with Wasm not working in Safari. The reason for this is that we used *WebAssembly.instantiateStreaming()* to instantiate our Wasm module. This function is the most efficient and optimised way to load a .wasm file to browsers, but Safari does not support this.

The reason for choosing *WebAssembly.InstantiateStreaming()* is that it is the most efficient way of fetching and instantiating Wasm modules as it does not require conversion to an ArrayBuffer [39]. However, by using a polyfill to check if the browser does not support this function, we can use the old version instead. With this polyfill, WasmStuff now runs in Safari too. Listing 6.1 shows how the polyfilling is done.

Listing 6.1: Polyfill to ensure WebAssembly is instantiated in all browsers

```
if (!WebAssembly.instantiateStreaming) {
        WebAssembly.instantiateStreaming = async (resp, importObject) => {
            const source = await (await resp).arrayBuffer();
            return await WebAssembly.instantiate(source, importObject);
        };
}
WebAssembly.instantiateStreaming(fetch("server.wasm"), go.importObject,{
    js: { mem: read , mem: write} }).then(result => {
        mod = result.module;
        inst = result.instance;
    }
 );
```

Platform	relab/hotstuff	WasmStuff
Mobile browser	✗	✓
Browser	✗	✓
Windows	✓	✓
MacOS	✓	✓
Linux	✓	✓

Table 6.4: WasmStuff execution environments compared to relab/hotstuff

In Table 6.4 we have provided an overview of the compatible execution environments for WasmStuff compared to those of relab/hotstuff. The only difference here is found when looking at browsers, both desktop and mobile. We ran some quick tests on mobile browsers for Android and iOS to see whether or not WasmStuff was compatible with them. The compatible mobile browsers are displayed in Table 6.5.

Mobile browser	Android	iOS
Chrome	✓	✓
Firefox	✓	✓
Opera	✓	✓
Edge	✓	✓
Safari	N/A	✓
Samsung Internet	✓	N/A

Table 6.5: WasmStuff compatibility on mobile browsers

Chapter 7

Discussion

Based on the results when comparing WasmStuff to the relab/hotstuff implementation, Figure 6.7, WasmStuff suffers no significant performance loss. Hence, we can conclude that our conversion to Wasm was a success. We achieved significant diversification while still retaining comparable performance on equal platforms as relab/hotstuff. Our modification and runtime environment allows for execution on nearly all available browsers. According to data gathered by Statcounter [40], WasmStuff has a 95,45% browser market coverage. These data are aggregated for both desktop and mobile users. The provided stats is an analysis of May 2021 from over 10 billion webpage views gathered from over 2 million websites. Details about the shares of individual browsers covered by WasmStuff can be seen in Figure 7.1.

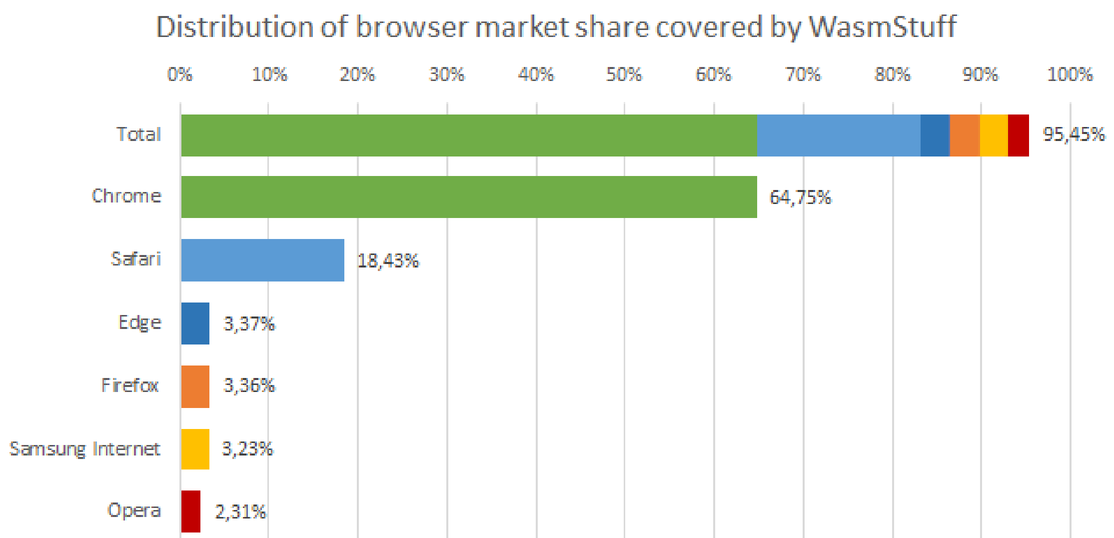


Figure 7.1: Market share of browsers supported by WasmStuff.

Another great thing that WasmStuff brings with its networking modifications is the ability to use it on pretty much any internet connected network without having to think about public IP or port forwarding for the replicas. As long as the web server and WS server is reachable, the replicas are good to go. WasmStuff has brought both extensive diversity and accessibility to BFT systems. In addition, the runtime provided by this thesis can be adapted to other BFT protocols, making it easy to set up different BFT protocols to be compilable to Wasm. Although this is possible, the runtime of WasmStuff could have been more modular. A more modular runtime would have lead to an even easier adaptation with fewer modifications necessary.

When researching different options for networking with Wasm, we opted for a solution that requires a web server to serve the webpage and for exchanging WebRTC SDP messages. Having a web server works well for simple local demonstration or geographically close public systems. In contrast, when deploying for worldwide systems, the distances become significant, and loading speeds increase. Using a single web server would also leave the system vulnerable due to having a single point of failure for the SDP message exchange.

A solution to this single point of failure is to deploy the server using a Content Delivery Network (CDN). CDNs provide distributed servers networked together to allow for shorter loading times all over the world. A CDN is an excellent choice for a large system running in active production. For our demonstration system, a single server was sufficient. By performing a small change to our current implementation, WasmStuff can provide rapid deployment of new web servers on the go. This change would make it possible to reconfigure the server IP on the go. A more detailed explanation is given in Section 8.2. By allowing simple IP reconfiguration, the single point of failure vulnerability is alleviated to some degree. All in all, with the rapid deployment design of browser-based WasmStuff, a CDN might prove a bit excessive.

Achieving diversity of BFT protocols has been a goal for many for quite some time now. As we introduced in Section 2.3, we found the Lazarus project, which also seeks diversity in BFT protocols. They chose the direction of a management system that replaced replicas based on vulnerabilities in their operating systems. WasmStuff, on the other hand, focuses on the direct ability to execute a single BFT implementation on a wide range of platforms. These two systems are not directly comparable, nor are they in competition with each other. In our opinion, the Lazarus management system is a fantastic tool to manage a diverse BFT system, but it does not provide diversity itself. They manage a range of operating systems on which they deploy a Java Virtual Machine to execute their chosen BFT protocol.

We built WasmStuff with Wasm and diversity in mind. With the use of Wasm, WasmStuff provides an easy to use runtime that can run on a wide assortment of platforms and machines without the added overhead of virtualization. With some modifications, these two systems could work together to provide both the diversity and the management for a BFT system.

As mentioned, diversity has been achieved through implementing a BFT protocol in browsers with Wasm. There are some flaws and omissions when looking at aspects other than diversity. Restrictions regarding allowing persistent storage of the blockchain restrain the usefulness of WasmStuff. JS does not have to possibility to store data directly on the users' computer, forcing WasmStuff to keep the blockchain in memory. Having to keep the whole chain in memory inhibits WasmStuff from being used for long-term persistent BFT system needs.

The development team at Chrome is working on a very exciting new API called File System Access [41]. This API is allowing more straightforward file system access on websites outside of a sandboxed environment. By granting read and write privileges to the website, the API can access and repeatedly update files without repeatedly prompting the user. Unfortunately, this API is only partially supported on Chromium-based browsers as of now. Therefore, WasmStuff is restricted as a demo, showing what is possible with Wasm, including diversifying a BFT protocol.

Chapter 8

Future Work

Before using WasmStuff in a production environment, one must add additional security and redundancy features. The BFT protocol is a strong defence against failures and malicious replicas, but if the nodes have weak security, adversaries can easily cross the safety threshold. Our scope focused on increasing resilience through diversification and have not implemented many additional security features. In this chapter, we present some additional features we would like to have implemented in the future. Some features involve increased security, others additional system robustness.

8.1 Security Features

We have pre-generated the public and private keys for each of the four configured replicas in the demonstration system. The keys are built into the code and are accessible to each replica. With the built-in keys, anyone can act as whichever replica they wish. In the future, we wish to see proper key generation locally on each of the participating replicas. This addition would be reasonably simple to implement as the key generation code exist already. Since each participating replica generates their keys upon joining a system, they hold the only copies of the keys. These keys could be exchanged between the replicas using the established WebRTC connections. Using WebRTC would be an easy way of exchanging the keys, but it would not lead to increased security.

We can increase security by uploading the public keys to a trusted public key distribution server where their identities can be verified. With WasmStuff, it is straightforward to add the capability to allow the user to input the keys of the other users. Text box or terminal inputs would allow users to exchange their keys in their preferred way and manually input it on their replica.

When the participating replicas are verified and keys are exchanged, message encryption would be the next step. Implementing message encryption is a trivial matter. There are easy to use internal package in Go that can encrypt and decrypt data. Replicas can exchange encrypted messages using asymmetric encryption, also known as public-key encryption. Before sending a message, the replica takes the public key of the receiver and encrypts the message. The replica receiving the message uses their private key to decrypt the message. Now the protocol messages and content is secured from possible attackers listing on the connection.

8.2 General Improvements

WasmStuff does not currently support the reconnection of failed replicas. After the initial networking setup, the replicas are no longer attempting to establish new connections. The main reason for this is that WasmStuff does not have an implemented catch-up feature. A failed node that reconnected would not have the ability to rejoin the running system due to having an outdated blockchain. Both of these features are desired for the future development of WasmStuff.

The reconnection of a failed replica is not too hard to implement. Since it needs to connect to all the other replicas, it should take the role of the listening replica. When the other replicas notice that a peer has disconnected, they should go into dialling mode, creating offers and sending them to the WS server. Through this process, the rejoining replica will rapidly restore the connections to the other replicas. After the reestablishment of all the connections, the replica should send a catch-up request to another replica. The other replicas should answer the request and share the missing blocks and current view number with the rejoining replica. Implementing a marshalling and unmarshalling procedure will allow for easy conversion and transferring of the missing blocks. When the replica has caught up to the rest, it should activate its pacemaker and start participating in the protocol.

The current Wasm binary and the web server has the IP address of the host machine hardcoded. All files have to be recompiled with a new address to change the IP address of the host. Having to recompile to code is a weakness and goes against the rapid deployment idea of a browser-based BFT system. We would like to extend the current implementation to accommodate this by adding the ability to input the desired IP as a parameter when starting the web server. Likewise, on the Wasm binary, we would like to add a function that would reconfigure the IP stored for the WS connection. Not only would this allow rapid reconfiguration for system users, but it would also lessen the vulnerability caused by having a single point of failure.

The WebRTC protocol is reliant on a STUN server to detect the network address of the browser windows. A browser window does not know its own network address and needs to query STUN server to receive a response with their IP and port number. Our current implementation is configured to use one of Google's publicly available STUN servers. To increase the robustness and decrease the dependability of WasmStuff, we would like to prepare and package a STUN server together with the web server. Removing the dependency on goodwill services will increase system sustainability for the future.

When designing the networking for the test system, we configured it to handle four replicas. We wish to improve upon this in the future to allow easier expansion of the number of participating replicas. Due to the nature of our chosen communications protocol, WebRTC, connection establishing has to follow a strict procedure. Each end of a connection has to perform different steps, which is making this somewhat more challenging.

Our current logic has a predetermined amount of replicas participating. In that way, each replica knows what steps to perform at the different stages of the connection phase. These are the changes we think our current implementation would need to allow this. The replica with the lowest ID should start as the connection leader performing the listening role. All the other replicas should start performing the dialling role. When the connection leader has established connections to every replica, it should inform the replica with the next-lowest ID to start acting as the connection leader. By repeating this process, each replica should have established connections at the end.

8.3 Further Diversification

We highly desire the ability to use WASI for WasmStuff. When the use of WASI with Go becomes more developed and supported, we wish to modify the current code to take full advantage of the available resources. With WASI bridging the gap between sandbox and networking, we hope to bring WasmStuff to become genuinely peer-to-peer without the WS server for SDP message exchange. We can further the diversification by utilizing WASI and the possibility of using off the browser Wasm runtimes. With runtimes supporting multiple languages, the same WASI implementing Wasm code can efficiently run on almost all available computer systems. WASI is bringing diversification through both hardware and software platforms.

Chapter 9

Conclusion

The overall goal of this thesis was to take advantage of the new technology provided by Wasm and achieve diversity in BFT protocols. We propose WasmStuff, an adaptation of the relab/hotstuff implementation of HotStuff that is compiled to Wasm and runs in almost any browser. Through modifying and compiling a previous implementation of a BFT protocol to Wasm and deploying it in browsers, we achieved our goal. WasmStuff is, to our knowledge, the first implementation of a BFT protocol where the replicas are contained within a browser. Through research and analysis, we discovered different ways of achieving network connections in Wasm, opting for the only solution providing peer-to-peer connections.

Experimental evaluation proves that the conversion of relab/hotstuff's implementation to Wasm did not have a significant impact on the performance of WasmStuff. Furthermore, the experiments performed on multiple platforms and browsers substantiates the achieved diversity. The runtime created for WasmStuff can be modified to work with other BFT protocols, providing them with the same diversity. This thesis contributes to the field of computer science, more specifically to Wasm and BFT, with a well documented procedure of all the steps needed to create WasmStuff. With WasmStuff and the techniques described in this thesis, BFT State Machine Replication can be easy to use and available for anyone.

List of Figures

2.1 Illustration of how pipelining in HotStuff is done. 8

4.1 Illustration of how WebRTC connectivity is achieved. 22

5.1 Overview leader replica's runtime 35
5.2 Overview of normal replica's runtime 36

6.1 Performance of WasmStuff on four replicas in Chrome, with 500, 2000 and 5000 executed commands. 44
6.2 Performance of WasmStuff on four replicas in Firefox, with 500, 2000 and 5000 executed commands. 45
6.3 Performance of WasmStuff on four replicas in Opera, with 500, 2000 and 5000 executed commands. 45
6.4 Performance of WasmStuff on four replicas in Edge, with 500, 2000 and 5000 executed commands. 46
6.5 Comparison of execution times of all four browsers for 5000 executed commands. 46
6.6 Comparison of execution times of all four browsers and relab/hotstuff for 5000 executed commands. 47
6.7 Comparison of execution times of WasmStuff and relab/hotstuff for 5000 executed commands. 47
6.8 Performance of a mixed browser system with 500, 2000 and 5000 executed commands. 48
6.9 Performance of systems with different numbers of replicas running on either Windows or in a browser with 5000 executed commands. 49
6.10 Comparison of adjusted and unadjusted execution times with regards to timeouts. 50

7.1 Market share of browsers supported by WasmStuff. 53

A.1 Front page of WasmStuff 67
A.2 Option to start benchmark or to play chess 68
A.3 Benchmark of WasmStuff 69
A.4 Servers are connecting before an opponent can be chosen. 70
A.5 Chess game started. 70

List of Tables

5.1 Possible command phrases and included data of a WS server. 27
5.2 Methods in the consensus interface 32
5.3 Methods in the synchroniser interface 32
5.4 Struct of replica server 33

6.1 Average execution time for 5000 executed commands 48
6.2 Test combinations for a cross-platform system 49
6.3 WasmStuff browser compatibility 51
6.4 WasmStuff execution environments compared to relab/hotstuff 52
6.5 WasmStuff compatibility on mobile browsers 52

C.1 Chrome data 75
C.2 Firefox data 76
C.3 Opera data 77
C.4 Edge data 78
C.5 Windows data 79
C.6 Linux data 81
C.7 relab/hotstuff Windows data 82
C.8 Mixed browsers data 83
C.9 1 Windows + 3 Browsers data 84
C.10 2 Windows + 2 Browsers data 85
C.11 3 Windows + 1 Browser data 85
C.12 3 Windows + 1 Browser unadjusted data 86
C.13 Average execution times 87

Appendix A

User Guide

The complete code repository can be found on GitHub at WasmStuff. The easiest way to demo the system is to download the release from GitHub called *WasmStuff Initial Release*. The release contains all the necessary files to run and test the system locally.

1. Download the release named *WasmStuff Initial Release* from WasmStuff
2. Open a terminal and navigate to the folder *HotstuffWASM/newNetwork*
3. Start the web server by inputting this command *websocket/websocket.exe*
4. Go to http://127.0.0.1:8080/websocket/server.html

If you executed the file correctly from the *newNetwork* folder, below you can find a step-by-step tutorial on how to use/test WasmStuff:

Figure A.1: Front page of WasmStuff

1. Open WasmStuff in four browser windows/tabs in the browser of your choosing.
2. In the top left corner you can purge the WebRTC database or download an MD5 Checksum.
3. Set the ID of each server in the separate windows.
4. Choose whether to run the benchmark or to play chess.

Figure A.2: Option to start benchmark or to play chess

The following steps help start the benchmark option, see step 7-8 for how to start a game of chess.

4. Set the number of commands you would like to be executed. If omitted, default is set to 1000.
5. Click ***GO*** to start.
6. When executed commands start appearing, you are free to send your own commands using the input field.

Figure A.3: Benchmark of WasmStuff

The following steps help you start a game of chess:

Figure A.4: Servers are connecting before an opponent can be chosen.

7. Once the buttons are enabled, you can choose who you want to play against. This is only done in one of the windows/tabs.

8. The game is ready, and the player who was invited (white) can start with their first move.

Figure A.5: Chess game started.

Appendix B

Marshalling and Unmarshalling

B.1 Marshalling

Listing B.1: Marshaller for a NewView to string

```
1 func NewViewToString(view hotstuff.NewView) string {
2     msg := "NewView:" + strconv.FormatUint(uint64(view.ID), 10) + ":" +
3     strconv.FormatUint(uint64(view.View), 10) +":" +
4     view.QC.GetStringSignatures() + ":" +
5     view.QC.BlockHash().String()
6     return msg
7 }
```

Listing B.2: Marshaller for a Partial Certificate to a string

```
1 pcString := pc.GetStringSignature() + ":" + pc.BlockHash().String()
2
3 func (cert PartialCert) GetStringSignature() string {
4     return cert.Signature.ToString()
5 }
```

Listing B.3: Marshaller for a Block to string

```
1 func (b *Block) ToString() string {
2     block := b.hash.String() + ":" +
3     b.Parent.String() + ":" +
4     strconv.FormatUint(uint64(b.Proposer), 10) + ":" +
5     string(b.Cmd) + ":" + b.Cert.GetStringSignatures() + ":" +
6     b.Cert.BlockHash().String() + ":" +
7     strconv.FormatUint(uint64(b.View), 10)
8     return block
9 }
```

B.2 Unmarshalling

Listing B.4: Unmarshaller for a string to a NewView

```
func StringToNewView(s string) hotstuff.NewView {
    stringByte := strings.Split(s, ":")
    viewID, _ := strconv.ParseUint(stringByte[1], 10, 32)

    view, _ := strconv.ParseUint(stringByte[2], 10, 32)

    certHash, _ := hex.DecodeString(stringByte[4])
    var c [32]byte
    copy(c[:], certHash)
    certHash2 := hotstuff.Hash(c)
    var sig map[hotstuff.ID]*hsecdsa.Signature
    sig = make(map[hotstuff.ID]*hsecdsa.Signature)
    sigString := stringByte[3]

    sigBytes := strings.Split(sigString, "\n")
    if sigString != "" {
        for i := 0; i < len(sigBytes)-1; i++ {
            m := strings.Split(sigBytes[i], "=")
            id, _ := strconv.ParseUint(m[0], 10, 32)
            signString := strings.Split(m[1], "-")
            rInt := new(big.Int)
            rInt.SetString(signString[0], 0)
            sInt := new(big.Int)
            sInt.SetString(signString[1], 0)
            signer, _ := strconv.ParseUint(signString[2], 10, 32)
            sign := *hsecdsa.NewSignature(rInt, sInt, hotstuff.ID(signer))
            sig[hotstuff.ID(id)] = &sign
        }
    }

    var cert hotstuff.QuorumCert = hsecdsa.NewQuorumCert(sig, certHash2)

    newView := hotstuff.NewView{
                ID: hotstuff.ID(viewID),
                View: hotstuff.View(view),
                QC: cert,
                }
    return newView
}
```

Listing B.5: Unmarshaller for a string to a Block

```
1  func StringToBlock(s string) *hotstuff.Block {
2      strByte := strings.Split(s, ":")
3      parent, _ := hex.DecodeString(strByte[1])
4      var p [32]byte
5      copy(p[:], parent)
6      parent2 := hotstuff.Hash(p)
7      proposer, _ := strconv.ParseUint(strByte[2], 10, 32)
8      cmd := hotstuff.Command(strByte[3])
9      certHash, _ := hex.DecodeString(strByte[5])
10     var c [32]byte
11     copy(c[:], certHash)
12     certHash2 := hotstuff.Hash(c)
13     var sig map[hotstuff.ID]*hsecdsa.Signature
14     sig = make(map[hotstuff.ID]*hsecdsa.Signature)
15     sigString := strByte[4]
16
17     sigBytes := strings.Split(sigString, "\n")
18     if sigString != "" {
19         for i := 0; i < len(sigBytes)-1; i++ {
20             m := strings.Split(sigBytes[i], "=")
21             id, _ := strconv.ParseUint(m[0], 10, 32)
22             signString := strings.Split(m[1], "-")
23             rInt := new(big.Int)
24             rInt.SetString(signString[0], 0)
25             sInt := new(big.Int)
26             sInt.SetString(signString[1], 0)
27             signer, _ := strconv.ParseUint(signString[2], 10, 32)
28             sign := *hsecdsa.NewSignature(rInt, sInt, hotstuff.ID(signer))
29             sig[hotstuff.ID(id)] = &sign
30         }
31     }
32
33     var cert hotstuff.QuorumCert = hsecdsa.NewQuorumCert(sig, certHash2)
34     view, _ := strconv.ParseUint(strByte[6], 10, 64)
35
36     b := &hotstuff.Block{
37         Parent:   parent2,
38         Proposer: hotstuff.ID(proposer),
39         Cmd:      cmd,
40         Cert:     cert,
41         View:     hotstuff.View(view),
42     }
43     b.Hash()
44     return b
45 }
```

Listing B.6: Unmarshalls a string into ID, type of message and content of message

```
func FormatBytes(msg []byte) (id hotstuff.ID, cmd string, obj string) {
    if len(msg) != 0 {
        msgString := string(msg)
        msgStringByte := strings.Split(msgString, ";")
        if len(msgStringByte) == 1 {
            return hotstuff.ID(0), "", ""
        }

        idString, _ := strconv.ParseUint(msgStringByte[1], 10, 32)
        id = hotstuff.ID(idString)

        cmd = msgStringByte[2]

        obj = msgStringByte[3]

        return id, cmd, obj
    }
    return hotstuff.ID(0), "", ""
}
```

Appendix C

Raw Benchmark Data

Table C.1: Chrome data

500	2000		5000		
4.782575104	4.838025088	4.8842	4.713499968	4.774425088	4.803324992
4.804499968	4.894174912	4.870450112	4.842549888	4.907174912	4.739749888
4.751674944	4.978075008	4.8818	4.743425024	4.81092512	4.676
4.759974976	4.920049984	4.858850112	4.9126	4.698449984	4.814850048
4.65397504	4.951625024	4.848424832	4.664350016	4.764275008	4.687849984
4.895300032	4.828174912	4.871225088	4.817225024	4.683699904	4.778249984
4.86802496	4.861825088	4.876249984	4.86782496	4.757450048	4.762150016
4.8822	4.864724992	4.846424896	4.759524992	4.562524992	4.80210016
4.86015008	4.84544992	4.866025088	4.90357504	4.636200064	4.867549888
4.91684992	4.904499968	4.769625024	4.908799936	4.741649792	4.81517504
	4.858225088	4.812174912	4.878475072	4.839450112	4.745550016
	4.86682496	4.813950016	4.752149952	4.76184992	4.87284992
	4.838500096	4.760475072	4.905475072	4.67655008	4.778350016
	4.852299968	4.933899904	4.903324992	4.627524992	4.622650048
	4.881049984	4.808900096	4.936649984	4.748874944	4.802449984
	4.845799936	4.877025024	4.990399872	4.75707488	4.837100096
	4.847775168	4.898424896	4.938875072	4.652300032	4.722499904
	4.85102496	4.851875072	4.959499904	4.700700096	4.861325056
	4.83667488	4.874974976	4.837600064	4.683949888	4.869749952
	4.83317504	4.875649984	4.972750016	4.75095008	4.803924928
			4.958624896	4.818125056	4.7894752
			5.030200064	4.688974912	4.80987488

Continued on next page

Table C.1 – continued from previous page

500	2000		5000		
			4.890500096	4.773474944	4.90172512
			4.874675008	4.811550016	4.659924992
			4.808824896	4.797924928	4.845974912
			4.831650112	4.75957504	4.800100096
			4.756974976	4.727925056	4.89824992
			4.755324928	4.70302496	4.84104992
			4.716575168	4.866849984	4.794775104
			4.727350016	4.742174976	4.85697504
			4.705449984	4.76337504	4.862625088
			4.641249984	4.738899968	4.85055008
			4.62562496	4.788475008	4.7976
					4.83177504

Table C.2: Firefox data

500	2000		5000		
3.298	3.584249984	3.254	3.927999936	3.292749952	3.18024992
3.342750016	3.227	3.171500032	3.306750016	3.212750016	3.096500032
3.379749952	3.203750016	3.214999936	3.203	3.26975008	3.147750016
3.25075008	3.348499968	3.236	3.316249984	3.353749952	3.147499968
3.399749888	3.240500032	3.239	3.254749952	3.408750016	3.301749952
3.346500032	3.109249984	3.269	3.236	3.361750016	3.247750016
3.275000064	3.262	3.305	3.267000064	3.300499968	3.125000064
3.221999936	3.159	3.266750016	3.340750016	3.265750016	3.102999936
3.227	3.178000064	3.403250048	3.248249984	3.22824992	3.24275008
3.327	3.099999936	3.196	3.202	3.433000064	3.343499904
	3.442499968	3.267999936	3.303250048	3.112000064	3.141
	3.358750016	3.326499968	3.358249984	3.263749952	3.07375008
	3.267749952	3.155249984	3.353749952	3.343249984	3.287999936
	3.268000064	3.173250048	3.308249984	3.288750016	3.260000064
	3.313500032	3.146750016	3.291000064	3.434499968	3.250749952
	3.358499968	3.160500032	3.300750016	3.20475008	3.127499968
	3.313500096	3.313499968	3.31824992	3.280249984	3.022250048

Continued on next page

Table C.2 – continued from previous page

500	2000		5000		
	3.371499904	3.278000064	3.291250048	3.186250048	3.152500032
	3.335249984	3.279999936	3.31575008	3.223499904	3.165
	3.323000064	3.266	3.422	3.26975008	3.304
			3.369499968	3.302749888	3.298749952
			3.251499968	3.336000128	3.206500032
			3.372500032	3.271999936	3.129249984
			3.170749952	3.404499968	3.168750016
			3.305750016	3.203000064	3.251000064
			3.313	3.227499968	3.302250048
			3.250499904	3.371500032	3.252999936
			3.316250112	3.398749888	3.214249984
			3.301749888	3.284250112	3.258500032
			3.266500032	3.293500032	3.326
			3.35375008	3.221	3.234749952
			3.294249984	3.338	3.345500032
			3.182999936	3.242750016	3.229250048
					3.323499968

Table C.3: Opera data

500	2000		5000		
5.158295104	4.523448832	5.103685056	5.06428864	5.34484992	5.329683776
5.25863616	4.595555072	5.044228736	5.168775168	5.321231232	5.439915008
5.321910016	4.608744896	5.105349952	5.273338752	5.346700032	5.430491328
5.34860864	4.751327488	5.12386624	5.362111232	5.324533824	5.4269376
5.264791232	4.548816192	5.063032576	5.327758656	5.2008688	5.423418688
5.26912	4.568378816	5.056746304	5.308740032	5.358207552	5.39624
5.259039936	4.638417536	5.07550496	5.273731328	5.207276288	5.37004256
5.2396688	4.639926272	4.985068736	5.326083712	5.114851264	5.438012416
5.282536256	4.568830016	5.138422464	5.37006752	5.173937408	5.392760064
5.24367872	4.523247552	5.001193728	5.49609632	5.408260032	5.318997504
	4.533589888	5.05383744	5.37170752	5.28048	5.27106752
	4.610796096	5.039443776	5.353210048	5.274156288	5.23115744

Continued on next page

Table C.3 – continued from previous page

500	2000		5000		
	4.645212416	5.00981376	5.352847488	5.40437248	5.171612544
	4.683858816	5.119256256	5.211984896	5.392708736	5.292718656
	4.557003712	5.021628736	5.254223872	5.372633792	5.155478784
	4.48096992	5.016351232	5.425931328	5.364033728	5.255468672
	4.509093696	5.034153664	5.247857408	5.376976192	5.391513856
	4.686966208	5.085633792	5.347250112	5.332782592	5.373414976
	4.568528768	5.104386304	5.374619968	5.315056128	5.304532608
	5.129665088	5.048717568	5.300306304	5.302328704	5.34082624
			5.378418752	5.33481376	5.27177504
			5.296300096	5.360648768	5.413853696
			5.329654912	5.393101248	5.338432576
			5.315871232	5.29460128	5.411331264
			5.335396224	5.2737888	5.356286336
			5.349378752	5.466447552	5.492139968
			5.195208768	5.375187584	5.321862464
			5.303643648	5.465002496	5.275534976
			5.332240064	5.382035072	5.320932608
			5.31741632	5.3537824	5.257237376
			5.341017472	5.381458752	5.311888704
			5.372176192	5.419750016	5.276847424
			5.30707136	5.290355136	5.376054912
					5.348708736

Table C.4: Edge data

500	2000		5000		
4.747800064	4.908750016	4.868124992	4.175924992	4.484150016	4.101549952
4.91502496	5.039774976	4.90902496	4.230499904	4.387600064	4.270249984
4.842950016	4.928875008	4.905500032	4.408950144	4.523349952	4.223324992
4.87937504	4.88182496	4.90157504	4.507499968	4.34404992	4.1458
4.838475008	4.834400128	4.874649984	4.3994	4.375275072	4.212875008
4.816075008	4.854974976	4.883724992	4.449574976	4.464924992	4.280199936
4.859549952	4.887174912	4.941250048	4.559575104	4.208650048	4.284125184

Continued on next page

Table C.4 – continued from previous page

500	2000		5000		
4.870325056	4.858675072	4.895649984	4.322074944	4.336275008	4.187624896
4.842325056	4.823475008	4.913124992	4.43947488	4.489550016	4.264650048
4.822425024	4.93035008	4.958950016	4.372200064	4.453124928	4.297350016
	4.893449856	4.930850048	4.387250048	4.468150016	4.288675008
	4.895100032	4.921475008	4.458250048	4.398650048	4.241600064
	4.895824896	4.860950016	4.326274944	4.350250048	4.19462496
	4.935100096	4.967400064	4.356000064	4.412024896	4.223475008
	4.930850048	4.874749952	4.549074944	4.460725056	4.186075008
	4.910050048	4.939899968	4.5036	4.30002496	4.365149952
	4.8918	4.959199936	4.392750016	4.257025088	4.200699968
	4.891174848	4.936725184	4.48077504	4.309099968	4.301675008
	4.86997504	4.91962496	4.388750016	4.456574976	4.205825024
	4.905425088	5.01669984	4.326049984	4.306375104	4.350224896
			4.324050112	4.231849792	4.088175104
			4.385774912	4.221600064	4.15482496
			4.352225024	4.2688	4.18657504
			4.482075008	4.23795008	4.210750016
			4.257275008	4.2872	4.094600064
			4.430800064	4.184700032	4.232899904
			4.410949888	4.275524992	4.203375104
			4.564699904	4.319500032	4.163399936
			4.507900224	4.202849856	4.08635008
			4.583549824	4.097900096	4.10902496
			4.375800064	4.070699968	4.17764992
			4.411100032	4.082100032	4.098700032
			4.423150016	4.044950016	4.192050112
					4.17884992

Table C.5: Windows data

500	2000		5000		
0.12745235	0.1400268	0.1178013	0.1128185	0.1347944	0.1617128
0.1405831	0.1025705	0.113913	0.1051472	0.1454691	0.1370383

Continued on next page

Table C.5 – continued from previous page

500	2000		5000		
0.1581508	0.1050432	0.0921063	0.1479266	0.093188	0.169281
0.14420625	0.098632	0.1530711	0.1909098	0.0980568	0.1376226
0.15381085	0.1326331	0.1493436	0.1421545	0.1451821	0.133487
0.16562975	0.1399654	0.1345904	0.126683	0.1485117	0.1338873
0.13283595	0.1351904	0.1316046	0.1429982	0.134559	0.1341267
0.133796	0.1285893	0.1553663	0.1481791	0.1375088	0.1716091
0.12138365	0.1602385	0.1458694	0.1839708	0.1469445	0.1411285
0.1195344	0.1420102	0.1296705	0.15375	0.1175654	0.1781563
	0.1431375	0.1149779	0.167317	0.092818	0.1480788
	0.147434	0.1161583	0.1408976	0.0938127	0.1450935
	0.1314091	0.105951	0.1354399	0.0919554	0.1586376
	0.153765	0.0954573	0.1794061	0.1301302	0.1399634
	0.1330112	0.1008957	0.1523335	0.1406207	0.1663317
	0.1204584	0.1799217	0.1246084	0.1414219	0.1562484
	0.1426874	0.1534841	0.1980549	0.1517436	0.1440747
	0.1824588	0.1533642	0.142438	0.1807404	0.1454552
	0.1250257	0.152352	0.1470287	0.1435198	0.1449791
	0.1153835	0.1456338	0.1170491	0.1243027	0.1561142
			0.1201325	0.142544	0.1606868
			0.1375532	0.125711	0.1166655
			0.104744	0.132067	0.1173204
			0.0922753	0.111069	0.1342274
			0.1010316	0.1523759	0.1438067
			0.1262181	0.1803204	0.1557688
			0.1321814	0.1327552	0.1350853
			0.1308473	0.1476488	0.1250284
			0.1297585	0.1425749	0.1734114
			0.1457589	0.1340011	0.1202675
			0.1531427	0.1347439	0.1223803
			0.1375795	0.16074	0.1141704
			0.1438246	0.1201863	0.1188268
					0.1259457

Table C.6: Linux data

500	2000		5000		
0.261449691	0.255937655	0.261656708	0.257017194	0.260664983	0.25967503
0.260007102	0.257118542	0.255844829	0.258512473	0.256576839	0.259041857
0.256427054	0.259914118	0.258577468	0.260639331	0.261410719	0.259491596
0.261645522	0.2582293	0.256799149	0.260109307	0.260884138	0.260237975
0.259631454	0.259207529	0.257214245	0.257485143	0.259816568	0.258145927
0.258465893	0.257612238	0.259634249	0.257951783	0.25856466	0.259071629
0.259397087	0.25973682	0.261282717	0.261524941	0.260555239	0.259121854
0.259855559	0.258308977	0.258036548	0.259031145	0.258096238	0.260902192
0.258325765	0.257757144	0.259528614	0.258583719	0.259093266	0.257276858
0.259237805	0.258626324	0.259692761	0.261970044	0.25890777	0.259353712
	0.259194138	0.263426967	0.258823611	0.261152432	0.258924502
	0.255103348	0.256186627	0.261016186	0.259085307	0.258850354
	0.259700196	0.260253315	0.257501074	0.25994537	0.257419531
	0.260850191	0.261980269	0.258713085	0.256507175	0.264546435
	0.258167771	0.258801401	0.259080266	0.256630083	0.258524588
	0.258434141	0.259003273	0.259081887	0.258992029	0.259283686
	0.259373029	0.259004995	0.260043572	0.261449547	0.261727674
	0.260055248	0.257969206	0.258534241	0.261387849	0.261692214
	0.257195381	0.259563335	0.260358261	0.258639356	0.261642214
	0.259407021	0.258804908	0.260024154	0.256237364	0.258865088
			0.258693433	0.261991655	0.259532265
			0.258855622	0.258712578	0.25684744
			0.25688821	0.259106139	0.257776884
			0.259917551	0.260707949	0.260311215
			0.254997605	0.257204314	0.2588253
			0.261756653	0.258491431	0.26098616
			0.259832244	0.26182112	0.259693648
			0.257700724	0.258522068	0.260761949
			0.259868009	0.257373239	0.259326122
			0.260887259	0.260403789	0.258373596
			0.257666646	0.257431576	0.257265553
			0.257768964	0.258487094	0.258132454
			0.260787672	0.256267547	0.262832227
					0.258522408

Table C.7: relab/hotstuff Windows data

500	2000		5000		
0.085129975	0.0652391	0.06676565	0.0688832	0.0685303	0.069609825
0.0941022	0.093459925	0.0735753	0.082821225	0.0692427	0.07698845
0.099429975	0.090006925	0.08263905	0.086708175	0.064452925	0.0695181
0.127076525	0.067451625	0.0687611	0.065684325	0.063776575	0.0738378
0.152468325	0.08635315	0.0770013	0.074806825	0.0663812	0.064922325
0.131781875	0.125673425	0.0752678	0.0820697	0.0840723	0.071180375
0.161197025	0.07676615	0.07261015	0.07763375	0.071425525	0.068013525
0.131754825	0.074052025	0.0761192	0.06756915	0.09597275	0.082642975
0.152772075	0.074212725	0.0731705	0.063985	0.06799655	0.06423825
0.121707725	0.06555865	0.08734475	0.074485375	0.07184495	0.0810217
	0.0881943	0.067969175	0.100694375	0.073452225	0.065321625
	0.07566775	0.0744166	0.085658525	0.07306095	0.0648826
	0.07047765	0.077510775	0.069597975	0.0888899	0.0648632
	0.06551185	0.0773337	0.068439475	0.07599585	0.076067525
	0.07295385	0.07160245	0.072258975	0.0677099	0.071887825
	0.071980375	0.0628312	0.068042925	0.0723286	0.079936975
	0.065098875	0.08019015	0.080644575	0.065729525	0.0728648
	0.0682382	0.071869225	0.0848086	0.070856125	0.0700124
	0.0821036	0.071865825	0.074475025	0.0646618	0.0643288
	0.06826195	0.071101275	0.066073675	0.06685055	0.065475275
			0.071105125	0.0748478	0.063726925
			0.0730192	0.0705931	0.082195575
			0.067614275	0.068645175	0.072583775
			0.0758575	0.069214725	0.072266275
			0.0711462	0.0716394	0.073831075
			0.072709325	0.0641287	0.0702494
			0.0678563	0.0772715	0.062943075
			0.065997525	0.079711575	0.0634214
			0.06716315	0.079271525	0.063400525
			0.07295665	0.067907	0.07632795
			0.076241325	0.063662175	0.064678275
			0.074040875	0.064680825	0.079158575

Continued on next page

Table C.7 – continued from previous page

500	2000		5000		
			0.0797402	0.072586725	0.103246475
					0.28358095

Table C.8: Mixed browsers data

500	2000		5000		
3.860625088	3.7834	4.468750016	4.52042496	4.54235008	4.54670016
3.774949888	3.695699968	4.44624992	4.53462496	4.542449984	4.605799808
4.198175104	3.578024896	4.180050048	4.57182496	4.538349952	4.656300096
4.561774912	3.792625024	4.19335008	4.626500096	4.555299968	4.632849984
4.405974976	3.659650048	4.159149952	4.60364992	4.5196	4.510750016
4.373975104	3.68077504	4.145750016	4.5588	4.481150016	4.561100096
4.382149888	3.617999872	4.097625024	4.577925056	4.528324928	4.477174976
4.410500032	3.655800128	4.208274944	4.578874944	4.54875008	4.561824896
4.360925056	3.992700032	4.22915008	4.512249984	4.565675008	4.613025088
4.325724992	4.202574976	4.19884992	4.537724992	4.538549952	4.525724992
	4.2122	4.183324992	4.472099968	4.496124992	4.51969984
	4.342800064	4.223150016	4.565150016	4.546675008	4.563975168
	4.166100032	4.157599936	4.495450048	4.561074944	4.498074944
	4.14667488	4.178850112	4.5786	4.48017504	4.525374848
	4.129650112	4.175924928	4.506350016	4.517924992	4.400175104
	4.16257504	4.166950016	4.543774976	4.145974976	4.521475008
	4.228574976	4.164099968	4.513074944	4.652900032	4.446225024
	4.26804992	4.149649984	4.541399872	4.509099968	4.52049984
	4.208250048	4.129400128	4.516975168	4.518900032	4.47855008
	4.431400064	4.1906	4.576475072	4.312500032	4.58275008
			4.505349888	4.377749952	4.494374976
			4.616149952	4.417300032	4.543675008
			4.578925056	4.498250048	4.567500096
			4.570075072	4.54082496	4.515174912
			4.60489984	4.479774976	4.49277504
			4.619025024	4.321100032	4.538475008
			4.578700096	4.50047488	4.453275072

Continued on next page

Table C.8 – continued from previous page

500	2000		5000		
			4.5506	4.429850048	4.537724992
			4.537025024	4.511925056	4.528499904
			4.530374912	4.445750016	4.477
			4.478774976	4.439950016	4.520174976
			4.576750016	4.513324992	4.539350016
			4.48524992	4.466199936	4.506174976
					4.610025024

Table C.9: 1 Windows + 3 Browsers data

5000				
3.823473837	3.952922226	3.964653726	3.974344782	3.744477514
3.783009075	4.041472508	4.059204734	4.139038161	3.867649366
3.778152962	4.06307414	3.985891354	3.889574627	3.895000621
4.050090324	4.078231911	4.047228874	3.925529663	3.769066048
3.991944209	3.996399456	3.955654686	3.870880824	3.778752418
4.035008689	4.061154245	3.989306717	3.969787835	3.843635658
3.915572542	4.07728504	3.966439518	3.89204346	4.29617815
4.026919855	4.038550242	3.999125383	3.964243698	4.050695969
4.004607781	3.919672599	3.984646634	3.872653123	3.809327438
4.042050641	4.095975417	4.138119055	3.975587949	3.881389507
3.950481631	3.957924132	4.025675319	3.91224447	3.864183941
3.960825163	4.061459531	4.162920816	3.994947641	3.7969098
4.085055834	3.965213675	3.975787459	3.714278092	3.792751511
4.037627958	4.061498553	3.965252276	3.747492663	3.795743072
4.092077724	3.989911749	4.013044476	3.757479712	4.004004712
4.092220861	4.02907517	3.922178201	3.825488765	3.882983526
4.020544416	4.019039737	3.999004274	3.756279384	3.903483866
4.085512262	4.094758161	4.076263102	3.873124796	3.86876216
3.973266899	4.033075958	4.008841174	3.745191403	3.884286292
4.080029532	4.037454549	4.051905535	3.757786525	3.937436476

Table C.10: 2 Windows + 2 Browsers data

5000				
2.949654434	3.238187349	3.023179584	3.060491343	2.9011171
3.181556205	3.095259927	3.317629006	3.029062352	2.942835109
3.107686955	2.965465989	3.115700891	3.098022441	2.884257892
3.193406551	3.146588834	3.003529294	3.02040994	3.091733434
3.345298867	3.122224392	3.191084845	3.015919328	3.025764405
3.165457867	3.067840958	3.10462219	2.976957072	2.952448301
3.085049823	3.011102357	3.108204416	3.079613435	2.823125834
3.089230003	2.989664191	3.075308601	3.022779531	2.928973659
3.213881709	3.101158043	3.140381917	3.050887127	2.936828713
3.089300633	3.040754691	2.981796373	3.09986489	2.87044231
3.151815533	3.139534669	3.072301184	3.236863576	2.924858082
2.997232751	3.141700266	3.039289217	2.949649532	3.004216286
3.160048976	3.140659334	3.083253085	3.139387809	3.143536358
3.072032508	3.064329592	3.258078056	2.89559036	2.848249098
3.304695216	3.102501725	3.013825258	2.913849281	3.119227603
3.09670828	3.099095723	3.128087008	2.877505119	3.044384145
3.088153919	2.995157885	3.11806835	2.889901817	2.903789102
3.088280918	2.870613323	3.096546449	2.990118942	2.879369634
3.032642956	2.962119895	3.183855293	2.828872547	2.932073248
3.108425851	3.020557347	3.105521431	3.188613458	3.276891567

Table C.11: 3 Windows + 1 Browser data

5000				
1.61484694	1.752626432	1.65844201	1.782893849	1.655672884
1.702177603	1.661669782	1.648902966	1.656139577	1.66356126
1.626019791	1.673437019	1.659819217	1.651270839	1.659352436
1.824968865	1.798590166	1.651275457	1.650531125	1.637217724
1.677069142	1.752710434	1.657713935	1.63849329	1.670184208
1.657394875	1.784119666	1.648731551	1.668435193	1.662918592
1.667730401	1.750815874	1.740819663	1.735183127	1.662078117
1.646418958	1.693197427	1.643346969	1.658775581	1.650222727
Continued on next page				

Table C.11 – continued from previous page

5000				
1.663723707	1.670384972	1.665023751	1.653192809	1.660771598
1.803169843	1.653443945	1.666374693	1.771701183	1.644963582
1.669299875	1.768456774	1.663661713	1.750946775	1.660783093
1.655619549	1.680336967	1.654142607	1.642731636	1.642319874
1.667617235	1.753637026	1.680956509	1.654679566	1.658990211
1.652547484	1.668804889	1.645122735	1.759903415	1.830037516
1.670126257	1.786454734	1.661611348	1.674749826	1.740360939
1.671960775	1.657875225	1.631021852	1.658386118	1.679581636
1.670476566	1.6507589	1.675828856	1.643806199	1.780795417
1.662508916	1.651242301	1.63900212	1.663442767	1.638792566
1.675955445	1.665263859	1.648677897	1.866499925	1.658243226
1.66993119	1.642159456	1.641423878	1.64806875	1.654487159

Table C.12: 3 Windows + 1 Browser unadjusted data

5000				
1.61484694	2.252626432	1.65844201	2.282893849	1.655672884
1.702177603	1.661669782	1.648902966	1.656139577	1.66356126
1.626019791	1.673437019	1.659819217	1.651270839	1.659352436
2.324968865	2.298590166	1.651275457	1.650531125	1.637217724
1.677069142	2.252710434	1.657713935	1.63849329	1.670184208
1.657394875	2.284119666	1.648731551	1.668435193	1.662918592
1.667730401	2.125815874	2.240819663	2.235183127	1.662078117
1.646418958	1.818197427	1.643346969	1.658775581	1.650222727
1.663723707	1.670384972	1.665023751	1.653192809	1.660771598
2.303169843	1.653443945	1.666374693	2.146701183	1.644963582
1.669299875	2.143456774	1.663661713	2.375946775	1.660783093
1.655619549	1.805336967	1.654142607	1.642731636	1.642319874
1.667617235	2.253637026	1.680956509	1.654679566	1.658990211
1.652547484	1.668804889	1.645122735	2.134903415	2.705037516
1.670126257	2.286454734	1.661611348	1.799749826	2.240360939
1.671960775	1.657875225	1.631021852	1.658386118	1.804581636
1.670476566	1.6507589	1.675828856	1.643806199	2.280795417

Continued on next page

Table C.12 – continued from previous page

5000				
1.662508916	1.651242301	1.63900212	1.663442767	1.638792566
1.675955445	1.665263859	1.648677897	2.866499925	1.658243226
1.66993119	1.642159456	1.641423878	1.64806875	1.654487159

Table C.13: Average execution times

	Average execution time		
	500	2000	5000
Chrome	4.817522502	4.861965002	4.792520751
Firefox	3.306849997	3.26719375	3.2720525
Opera	5.264628486	4.840067464	5.329191183
Edge	4.843432518	4.908654378	4.308577754
Mixed browser	4.265477504	4.10005688	4.522804247
Windows	0.13973831	0.133030062	0.138903588
Linux	0.259444293	0.258829767	0.259257216
relab/hotstuff Windows	0.125742053	0.075680182	0.074674045
1 Windows + 3 Browsers			3.959184801
2 Windows + 2 Browsers			3.057938207
3 Windows + 1 Browser			1.681886153
3 Windows + 1 Browser Unadjusted			1.791886153

Bibliography

[1] John Ingve Olsen and Hans Erik Frøyland. Implementing hotstuff with gorums. Bachelor's thesis, University of Stavanger, Faculty of Science and Technology, Stavanger, Norway, June 2020.

[2] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain, 2018.

[3] Byzantine fault tolerance explained, 12 2020. URL https://academy.binance.com/en/articles/byzantine-fault-tolerance-explained.

[4] Tormod Erevik Lea, Leander Jehl, and Hein Meling. Towards new abstractions for implementing quorum-based systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2380–2385, 2017. doi: 10.1109/ICDCS.2017.166.

[5] Miguel Garcia, Alysson Bessani, and Nuno Neves. Lazarus: Automatic management of diversity in bft systems. In *Middleware '19: Proceedings of the 20th International Middleware Conference*, pages 241–254, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-7040-0. doi: https://doi.org/10.1145/3361525.3361550.

[6] Virtualbox, 2019. URL https://www.virtualbox.org/.

[7] Webassembly, 2021. URL https://webassembly.org/.

[8] The llvm compiler infrastructure, 2021. URL https://llvm.org/.

[9] Tinygo - a go compiler for small places, 2021. URL https://tinygo.org/.

[10] Webassembly high-level goals - webassembly, 2021. URL https://webassembly.org/docs/high-level-goals/.

[11] Faq - webassembly, 2021. URL https://webassembly.org/docs/faq/.

[12] Use cases - webassembly, 2021. URL https://webassembly.org/docs/use-cases/.

[13] Compiler internals, 2021. URL https://tinygo.org/docs/concepts/compiler-internals/.

[14] Sergio Gusmeroli, Salvatore Piccione, and Domenico Rotondi. A capability-based security approach to manage access control in the internet of things. *Mathematical and Computer Modelling*, 58(5):1189–1205, 2013. ISSN 0895-7177. doi: https://doi.org/10.1016/j.mcm.2013.02.006. URL https://www.sciencedirect.com/science/article/pii/S089571771300054X. The Measurement of Undesirable Outputs: Models Development and Empirical Analyses and Advances in mobile, ubiquitous and cognitive computing.

[15] openat(2) - linux man page, 2021. URL https://linux.die.net/man/2/openat.

[16] Wasi libc, 2021. URL https://github.com/WebAssembly/wasi-libc.

[17] Dan Gohman. Wasi: Webassembly system interface, 2019. URL https://github.com/WebAssembly/WASI/blob/main/docs/WASI-overview.md.

[18] Aaron Turner. New projects, 2021. URL https://madewithwebassembly.com/new-projects.

[19] Bela Hullar. Webssh client, 2021. URL https://www.ssheasy.com/.

[20] Jordon Mears. How we're bringing google earth to the web, 2019. URL https://web.dev/earth-webassembly/.

[21] Get started with the ndk, 2020. URL https://developer.android.com/ndk/guides.

[22] Objective-c++, 2021. URL https://en.wikipedia.org/wiki/Objective-C#Objective-C++.

[23] Webgl: 2d and 3d graphics for the web, 2019. URL https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.

[24] Release notes for safari technology preview 114, 2020. URL https://webkit.org/blog/11300/release-notes-for-safari-technology-preview-114/.

[25] Christian Berger and Hans P. Reiser. WebBFT: Byzantine Fault Tolerance for Resilient Interactive Web Applications. In *18th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 1–17, Madrid, Spain, June 2018. doi: 10.1007/978-3-319-93767-0_1.

[26] M.G. Merideth, Arun Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-fault-tolerant middleware for web-service applications. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 131–140, 2005. doi: 10.1109/RELDIS.2005.28.

[27] Wenbing Zhao. Bft-ws: A byzantine fault tolerance framework for web services. In *2007 Eleventh International IEEE EDOC Conference Workshop*, pages 89–96, 2007. doi: 10.1109/EDOCW.2007.6.

[28] Alysson Bessani, João Sousa, and Eduardo Alchieri. State machine replication for the masses with bft-smart. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 355–362, 06 2014. doi: 10.1109/DSN.2014.43.

[29] gsoap user guide, 2020. URL https://www.genivia.com/doc/guide/html/index.html.

[30] Anmol Sethi. websocket, 2020. URL https://pkg.go.dev/nhooyr.io/websocket.

[31] Webrtc api, 2021. URL https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API.

[32] Uday Hiwarale. The anatomy of WebAssembly: Writing your first WebAssembly module using C (C++), 2020. URL https://medium.com/jspoint/the-anatomy-of-webassembly-writing-your-first-webassembly-module-using-c-c-d9ee18f7ac9b.

[33] Liftoff: a new baseline compiler for webassembly in v8, 2018. URL https://v8.dev/blog/liftoff.

[34] Wasmtime: Introduction, 2020. URL https://docs.wasmtime.dev/.

[35] wasmtime, 2021. URL https://pkg.go.dev/github.com/bytecodealliance/wasmtime-go#NewInstance.

[36] Pion webrtc, 2021. URL https://github.com/pion/webrtc.

[37] Chris Oakman. Documentation, 2021. URL https://chessboardjs.com/docs.

[38] Jeff Hlywa. Chess.js, 2021. URL https://github.com/jhlywa/chess.js.

[39] Webassembly.instantiate(), 2021. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/instantiate.

[40] Browser market share worldwide - may 2021, 2021. URL https://gs.statcounter.com/browser-market-share#monthly-202105-202105-bar.

[41] The file system access api: simplifying access to local files, 2021. URL https://web.dev/file-system-access/.