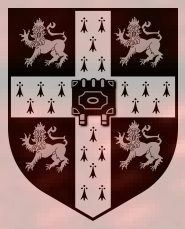
Marc Harvey-Hill

# Implementing the HotStuff consensus algorithm

Computer Science Tripos Part II
Gonville and Caius College
May 2023



# Declaration of Originality

I, Marc Harvey-Hill of Gonville and Caius College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: Marc Harvey-Hill

Date: 9$^{th}$ May 2023

# Proforma

| | |
|---|---|
| Candidate Number: | **2409B** |
| College: | **Gonville & Caius College** |
| Project Title: | **Implementing the HotStuff consensus algorithm** |
| Examination: | **Computer Science Tripos – Part II, 2023** |
| Word Count: | **11,612**[1] |
| Code line Count: | **4,190**[2] |
| Project Originator: | Christopher Jensen and 2409B |
| Supervisor: | Christopher Jensen |

## Original Aims of the Project

HotStuff is a byzantine-fault tolerant consensus algorithm that allows a group of participants to reach consensus when some proportion of them act maliciously. It can be used to develop blockchains which underlie cryptocurrencies and other decentralised applications. This project aimed to implement the HotStuff algorithm based on a paper by Yin et. al [1], demonstrate its correctness, and evaluate its performance. As an extension, further optimisations were to be implemented on the basic algorithm.

## Work Completed

This project includes a full specification of the HotStuff algorithm, filling in the gaps from the original paper with other protocols from the literature, and my changes. I have developed a correct implementation of the algorithm with several optimisations, which are shown to be effective in an evaluation. This project paves the way for future HotStuff implementations by describing non-trivial challenges that were encountered during development and giving solutions and practical optimisations. I synthesised information from the literature to give a full explanation of the algorithm and how it can be logically derived from additions to simpler consensus algorithms.

[1] Computed by `texcount -inc main.tex`
[2] Computed by *VS Code Counter*

## Special Difficulties

None.

# Contents

# Chapter 1

# Introduction

The promise of blockchains is to decentralise applications that were traditionally run in a centralised manner. The implications of this are far-reaching: central banks can be replaced by decentralised cryptocurrencies [2, 3], traditional corporations can be replaced with decentralised autonomous organisations (DAOs) [4, 5], internet infrastructure like DNS servers can be decentralised [6], and any possible algorithm can be run on a decentralised 'world computer' [5, 7]. The innovation that makes blockchains possible is the byzantine consensus algorithm.

Byzantine fault-tolerant consensus algorithms allow a group of parties to agree on some piece of information under adverse conditions where some messages can be lost and some parties are controlled by a malicious adversary. For example, one could create a cryptocurrency by using such an algorithm to reach consensus on a ledger of transactions like "Account Alice transfers account Bob £10"; the algorithm will ensure that transactions cannot be lost and the system cannot be sabotaged by malicious actors.

Byzantine consensus algorithms can be viewed as solutions to the byzantine generals problem [8], in which malicious actors are represented by byzantine generals. In this problem, a group of generals must all agree to siege a castle at the same time, but they can only communicate via messengers that take some time to arrive and can be captured en route. Additionally, up to a third of the generals may be malicious, and try to prevent the other generals from reaching consensus on a time to attack. By following a byzantine consensus protocol the generals can reach consensus on a value like "attack at dawn". Multi-valued consensus algorithms allow consensus to be reached on multiple values, resulting in a continuously growing log that can never be modified or erased, only extended. This problem statement assumes that the number of participating generals is fixed.

Blockchains can be either permissioned, or permissionless. Permissioned blockchains have a previously agreed set of participants in the consensus algorithm, whereas permissionless blockchains allow participants to join and leave freely. Most well-known blockchains, such as Bitcoin [2] and Ethereum [5, 7], are of the permissionless variety. Permissioned blockchains can be deployed in a permissionless setting if they are augmented with an additional layer of security, which can be proof of work, proof of stake, or some other similar mechanism. These aim to prevent a 'Sybil attack' where a large number of malicious nodes join the network, exceeding the threshold that only one third of nodes can be malicious. For example, proof of work adds a requirement for proof of computational work in order to participate in consensus, making Sybil attacks economically and computationally infeasible. Permissioned blockchains

are of interest for applications within a group or organisation, such as a company, where the participating nodes are known in advance.

HotStuff is a byzantine consensus algorithm that was notably used by Meta's Libra project [9], a cancelled permissioned blockchain-based payments system. The algorithm is relevant because of various performance advantages over other byzantine consensus algorithms such as PBFT [10], SBFT [11], DLS [12], Tendermint [13], and Casper [14].

Building practical, well-performing implementations of consensus algorithms is non-trivial. These algorithms are usually given in short pieces of pseudocode that may not be specified precisely and require much more code to implement in practice. Such software has a wide range of failure modes mostly due to their parallel nature, including deadlocks, resource starvation, and bugs in the implementation [15].

The main contributions of this dissertation are:

- Giving a complete specification and proof of HotStuff (Section 3.3), that adapts the pacemaker mechanism which was not specified in the original paper. This specification synthesises the chained algorithm described in the paper (Section 3.2.1), a view-change protocol based on other talks and papers (Section 3.2.2), and my changes to integrate the pacemaker with HotStuff without the need for synchronised clocks.
- Providing a reference implementation of HotStuff in OCaml based on a paper by Yin et. al [1].
- Giving solutions to key practical challenges of implementation and optimisations that can be made (Section 3.4), as well as showing their effectiveness (Section 4.3.3)
- Synthesising information from different sources to provide a complete explanation of the HotStuff algorithm, and how it can be arrived at through modifications to simpler consensus algorithms (basic algorithm Section 2.2, chained algorithm Section 3.2.1, pacemaker Section 3.2.2).

# Chapter 2

# Preparation

In this chapter I disclose my knowledge and experience prior to beginning this project (Section 2.1), give a theoretical basis for understanding the HotStuff algorithm by building up from simpler consensus algorithms (Section 2.2), outline the tools, libraries (Section 2.3), and professional methodology (Section 2.5) deployed in implementation, and highlight the requirements that the implementation should meet (Section 2.4).

## 2.1 Starting point

Although, I had some experience using OCaml through the IA Foundations of Computer Science course, I had never utilized it in a project before. The IB Distributed Systems course provided some useful background knowledge, particularly as it briefly covered Raft [16], a non-byzantine consensus algorithm. Additionally, I had a basic understanding of byzantine consensus from my own reading into Nakamoto consensus [2] and from developing a wallet application for Ethereum [5, 7]; neither of these was directly useful to implementing HotStuff, but they gave me some wider context of the field.

## 2.2 HotStuff algorithm

HotStuff is a byzantine consensus algorithm; it allows a group of nodes to reach consensus on a log of values under adverse conditions, such as messages being lost, or some nodes being byzantine. In each view a leader node proposes some value by sending it the replicas (another word for nodes). After several messages are exchanged some prefix of the log may be committed, meaning there is consensus on that part of the log and it is immutable.

The system model describes the adverse conditions under which HotStuff can operate:

**Assumption 2.2.1** (Partially Synchronous)**.** *Messages sent by one party will always be delivered to another within some bounded amount of time ($\Delta$) after global synchronisation time (GST) has been reached [12].*

**Assumption 2.2.2** (Authenticated)**.** *We assume that all messages are signed, providing an authenticated channel where no messages can be spoofed.*

**Assumption 2.2.3** (Byzantine)**.** *A maximum of $f$ faulty nodes may be controlled by an adversary that is actively trying to prevent the nodes from reaching consensus, where $n = 3f + 1$ and $n$ is the total number of nodes.*

Assumptions 2.2.1 and 2.2.3 are the weakest possible assumptions under which it is possible to reach consensus [17, 18]. Assumption 2.2.2 is also needed, but byzantine consensus is possible without cryptographic signatures; there is an algorithm called Information Theoretic HotStuff, which does not use signatures, and is secure against a computationally unbounded adversary[1] [20].

HotStuff has the following properties:

**Property 2.2.1** (Safety)**.** *Once some prefix of a log has been committed, that part of the log is immutable, it can only be appended to.*

**Property 2.2.2** (Liveness)**.** *Assuming there is a functioning pacemaker, the system is guaranteed to make progress within some bounded amount of time once GST has been reached and a non-faulty leader is chosen. The pacemaker ensures that all honest replicas will remain in some view with an honest leader for long enough to make progress (Section 3.2.2).*

**Property 2.2.3** (Optimistic Responsiveness)**.** *Once GST has been reached and a non-faulty leader is chosen, the system is able to make progress as fast as network conditions allow; it does not have to wait for some timeout to elapse [21].*

Basic HotStuff is a responsive byzantine consensus algorithm (Section 2.2.3). A simpler consensus algorithm that is neither responsive nor byzantine is described in Section 2.2.1. This algorithm is then extended in Section 2.2.2 to handle byzantine threats. Finally, Section 2.2.3 explains how the algorithm is made responsive by removing the need for a timeout to progress, arriving at the basic HotStuff algorithm.

### 2.2.1 Non-byzantine consensus

In this section, a generic algorithm is described to solve the problem of reaching consensus with a crash-stop model, which assumes that nodes cannot be malicious but can crash and never come back online. Examples of similar algorithms include Raft [16] and MultiPaxos [22, 23].

Each view is composed of several phases. In each phase, the leader broadcasts to the replicas, that respond with an acknowledgement (ack). The leader waits until it receives a quorum of acks before proceeding to the next phase; for this algorithm, a quorum consists of $\frac{n}{2}$ acks.

The final phase of a consensus algorithm is *decide*, it commences once the log is committed. This phase assumes that consensus is being used in the context of state machine replication [24, 25]: a client sends commands to the nodes, the nodes reach consensus on a log of commands, and then execute them in order, ensuring that all nodes will be in the same state.

[1] An authenticated channel can be obtained without signatures by using one time pads or quantum cryptography [19].

In the *decide* phase, the nodes can execute the new commands and respond to the client that the command was successfully committed.

In this generic algorithm, each view consists of three phases:

*new-view* — the leader learns about previously committed logs. All replicas send a *new-view* message to the leader, containing their longest previously committed log.

*commit* — the leader waits until it receives a quorum of greater than $\frac{n}{2}$ *new-view* messages, and then picks the longest log it received ($\lambda$) to propose. It then broadcasts a *commit* message to the replicas, proposing $\lambda'$ to the nodes, where $\lambda'$ is $\lambda$ optionally extended with the leader's new value.

*decide* — once the leader receives a quorum of acks, the log has been successfully committed. The leader can broadcast "decide" to the replicas, who can execute the new commands and respond to the client.

Crucially, this algorithm is safe (Property 2.2.1). Since the leader waits to receive a quorum of greater than $\frac{n}{2}$ *new-view* messages, $\lambda$ is guaranteed to be the longest log that has previously been committed. This is because the quorum of *new-view* messages must share at least one node with the past quorum of *commit* acks for $\lambda$. The new proposal $\lambda'$ will never conflict with $\lambda$, hence the algorithm is safe.

### 2.2.2 Byzantine consensus

In this section, consensus under a byzantine system model (assumption 2.2.3) is achieved by extending the non-byzantine algorithm introduced in Section 2.2.1. This is accomplished by first introducing the quorum certificate (QC), a cryptographic proof that a leader has received a quorum of acks. Next, the threats posed by byzantine nodes are considered, and a generic algorithm that solves these problems is presented, along with an argument for safety. Examples of similar algorithms include Tendermint [13] and Casper [14].

#### Quorum certificates

A QC is a quorum of $n - f$ acks with a matching threshold signature. A threshold signature combines several signatures of the same message into one [26, 27]; in this case, the signature from each ack is combined[2]. QCs have a key property that the byzantine algorithm will rely on:

**Property 2.2.4.** *There will always be at least one honest node in the intersection of any two QCs.*

Recall that $n = 3f + 1$. The property holds since a QC contains a quorum of $n - f$ acks, at least $f + 1$ of which must be from honest nodes. To have two quorums that do not share an

[2]Recall that messages are signed to provide authenticated delivery (assumption 2.2.2).

honest node would require at least $2(f+1) = 2f+2$ honest nodes, but the system only has $2f+1$.

### Threats introduced by byzantine nodes

Two threats are introduced by Byzantine nodes, which will be dealt with in turn. Solutions to these threats are presented; their effectiveness will become clear in the argument for the safety of the algorithm.

*Threat #1 (Equivocation)* — a faulty leader proposes one value to some replicas and a different value to others. For example, in the case of a cryptocurrency a malicious actor (Mallory) could carry out a double-spend attack by proposing "Mallory transfers Alice £10" to some nodes and "Mallory transfers Bob £10" to others, even if Mallory's account contains less than £20.

To solve this, add a new stage *prepare* which happens just before the *commit* phase, where the leader pre-proposes a log before proposing it in the *commit* phase.

*Threat #2* — a faulty leader proposes a log that conflicts with one that has previously been committed.

To solve this, make replicas *lock* on a proposal once they receive a *commit* message, and not accept a pre-proposal for a conflicting log.

### Byzantine fault-tolerant algorithm

Modifying the non-byzantine algorithm (Section 2.2.1) to include solutions to threats 2.2.2 and 2.2.2 results in the following:

*new-view — same as the non-byzantine algorithm.*

*prepare* — the leader waits $\Delta$ until it receives a *new-view* message from all replicas (this will be revisited in Section 2.2.3), and then picks the longest log it received ($\lambda$) to pre-propose. It then broadcasts a *prepare* message to the replicas, pre-proposing $\lambda'$ to the nodes, where $\lambda'$ is $\lambda$ optionally extended with the leader's new value. The replicas ensure that $\lambda'$ does not conflict with their *lock* before sending an ack.

*commit* — the leader waits until it receives a quorum of *prepare* acks. It then broadcasts a *commit* message to the replicas, proposing $\lambda'$ to the nodes, and includes a QC of *prepare* acks. The replicas then *lock* on $\lambda'$ and send a *commit* ack.

*decide — same as the non-byzantine algorithm.*

#### Argument for safety

An informal inductive argument is presented to show that the algorithm is safe (Property 2.2.1) based on it solving threats 2.2.2 and 2.2.2. Safety requires that if some log $\lambda$ is committed in view $v$, at no point in future will a conflicting log be committed.

Base case — in view $v$, no log that conflicts with $\lambda$ may be committed; in other words, equivocation (Threat 2.2.2) is not possible. For a view to commit a value, there must have been a QC of *prepare* acks, and a QC of *commit* acks. By Property 2.2.4, there must be at least one honest replica in the intersection of these QCs that would not have acknowledged conflicting proposals in the same view.

Inductive step — no conflicting log can be proposed in view $v'$ where $v' > v$ (Threat 2.2.2). This holds because any log pre-proposed in view $v'$ must receive a QC of *prepare* acks; by property 2.2.4, there must be at least one honest node in the intersection between this QC, and the QC of *commit* acks $\lambda$ in view $v$. This honest replica is locked on $\lambda$, so would not accept a proposal that conflicts with it.

From this, it follows that in no view from $v$ onwards will a log conflicting with $\lambda$ be committed, so the algorithm is safe.

### 2.2.3 Optimistic responsiveness

This section presents the basic HotStuff algorithm by extending the generic byzantine consensus algorithm (Section 2.2.2) to make it optimistically responsive (Property 2.2.3). Optimistic responsiveness means that the system can make progress as fast as network conditions allow without waiting for a timeout, once GST has been reached [21].

The byzantine consensus algorithm is not responsive as a timeout is needed in the *prepare* phase. The problem that necessitates this timeout is first described, followed by the presentation of an algorithm that solves it. An informal argument is made that liveness is not broken by this algorithm.

#### Why the timeout was necessary

For the system to have liveness (Property 2.2.2), the leader must wait for $\Delta$ to elapse so that it receives a *new-view* message from all honest replicas before it pre-proposes a log, to ensure that the pre-proposal will be voted for by the replicas. Consider what would happen if the leader did not wait for this timeout, and did not receive a *new-view* from some honest replica $x$. It is possible that $x$ is locked on a longer log than the other replicas, as in some past view it received the *commit* message, but other replicas did not[3]. When the leader sends the *prepare* message, $x$ will not vote the pre-proposal as it is locked on a longer log, so the leader will not acquire a quorum of acks to make progress; this breaks liveness.

[3]N.B. this means that the value was never actually committed, as this would have required a quorum of *commit* acks

Solution idea — add a *pre-commit* phase directly before the *commit* phase, where replicas store a *key* for a proposal that they include in their *new-view* message; this removes the need for a timeout, making the algorithm responsive.

#### Basic HotStuff algorithm

Modifying the byzantine algorithm (Section 2.2.2) to include the solution idea leads to the following:

*new-view* — all replicas send their *key* to the leader.

*prepare* — *same as the byzantine algorithm, but picks the key for the longest log.*

*pre-commit* — the leader waits until it receives a quorum of *prepare* acks. It then broadcasts a *pre-commit* message to the replicas, which contains a QC of *prepare* acks. The replicas store this QC as a *key* and send a *pre-commit* ack.

*commit* — *same as the byzantine algorithm, but creates QC from pre-commit acks instead of prepare acks.*

*decide* — *same as the non-byzantine algorithm.*

#### Argument for liveness

An informal argument for the liveness (Property 2.2.2) of the algorithm is presented. It is argued that the longest log will be proposed by the non-faulty leader, and that the proposal will be voted for by the honest replicas, resulting in progress being made.

The non-faulty leader is guaranteed to receive a *key* for the longest log ($\lambda^*$) that some honest replica is locked on. This is because for some replica to become locked on $\lambda^*$ there must be at least $f + 1$ honest nodes that have a *key* for $\lambda^*$. The leader must hear about $\lambda^*$ from one of these honest nodes when it receives a quorum of *new-view* messages and then propose it to the replicas.

The honest replicas will vote for the proposed log $\lambda^*$ as it does not conflict with their *lock* and they are in the same view. The honest replicas are guaranteed to be synchronised in the same view by the assumption that there is a functioning pacemaker (Section 3.2.2). Furthermore, the replicas will progress through the other phases by the assumption that GST has been reached and messages must be delivered within $\Delta$.

## 2.3 Tools & libraries

This section describes the languages and libraries used in implementation and justifies why they were appropriate for this project.

### 2.3.1 OCaml

I chose OCaml [28] for this project due to its high-level nature, static type system, ability to blend functional and imperative paradigms, powerful module system, and good library support. The performance bottlenecks for distributed byzantine algorithms are generally cryptography, message serialisation and network delays. This means that it is more important to choose a language with suitable features to aid implementation, rather than picking a 'high-performance' language like C++.

OCaml's multi-paradigm nature is suitable for implementing HotStuff, as the core state machine can be elegantly expressed in a functional way, whereas interacting with the RPC library to send messages is better suited to an imperative paradigm.

OCaml has a powerful module system that facilitates writing highly reusable code that was only briefly touched upon in the tripos (in Concepts in Programming Languages from IB). A modular design allows key components such as the consensus algorithm to be easily reused in other projects.

### 2.3.2 Lwt

Lwt [29] is a concurrent programming library for OCaml. It allows the creation of promises, which are values that will become determined in the future; these promises may spawn threads that perform computation and I/O in parallel. In order to use Lwt I had to learn about monads, which are ways of sequencing effects in functional languages that are used by promises in Lwt.

Lwt is useful to this project as promises provide a way to asynchronously dispatch messages over the network and wait for their responses in different threads. Promises are cheap to create in Lwt, so one can create many lightweight threads with good performance.

There are alternative libraries I could have used that may have had better performance, namely Jane Street's Async library [30] and EIO [31]. I chose Lwt over these libraries due to superior documentation and stability.

### 2.3.3 Cap'n Proto

Cap'n Proto [32] is an RPC framework that includes a library for sending and receiving RPCs, serialising messages, and a schema language for designing the format of RPCs that can be sent. Benchmarks for the library are presented in Section 4.2.1.

### 2.3.4 Tezos cryptography

The Tezos cryptography library [33] provides aggregate signatures using the BLS12-381 elliptic curve construction. It provides functions to sign some data using a private key, aggregate several signatures into a single one, and check whether an aggregate signature is valid. Benchmarks for the library are presented in Section 4.2.2.

The only difference from the threshold signatures needed by HotStuff is that each individual signature in an aggregate signature can sign different data, whereas with threshold signatures each individual signature is over the same data. It is trivial to implement threshold signatures using this library by checking that the data is the same for all signatures inside the aggregate signature.

## 2.4 Requirements analysis

In order to be successful the implementation should conform to the following requirements:

- Correctness — the consensus algorithm should be implemented as it is described in the paper [1]. This can be established by testing the program trace for compliance with the algorithm specification.
- Evaluation — analysis of system throughput and latency should be carried out by testing the program locally, analysing the trace, and testing in a simulated network.
- Optimisation — implement features to improve transaction throughput and reduce latency over the naive implementation. This can be achieved through architectural decisions, tuning the scheduler, and ensuring cryptographic libraries are being used efficiently.

## 2.5 Software engineering practices

This section describes the professional software engineering methodology deployed during implementation and justifies why this project is ethical.

### 2.5.1 Development methodology

For this project, I used an iterative waterfall development methodology. Objectives were chosen in accordance with the timetable set out in the proposal (Appendix A). Development then proceeded in cycles of implementation, testing, and analysis of the program trace (to compare it to the protocol specification) and timing statements. This approach was particularly useful during the optimisation of the system (Section 3.4), which involved extensive log analysis and rapid prototyping to compare performance.

### 2.5.2 Testing & debugging methodology

Unit testing was carried out using 'expect tests', which compare a program trace to the correct output. A testing suite of expect tests verifies that the program behaves as specified in the HotStuff paper. This suite has 100% code coverage[4] of the consensus state machine code, the coverage report is at *_coverage/index.html*.

The Memtrace library and viewer [34] were used to profile the memory usage of the program. One can generate a flame graph of memory allocations to see which parts of the program are using the most memory.

The Mininet virtual network [35, 36] was used to test the system in a simulated wide area network (Section 4.3.4).

The distributed nature of this project meant that debugging deadlocks and performance issues had to be carried out by manual inspection of the program trace and timing sections of the program. This is because the cause of these issues is often some process waiting or a backlog of work forming on some node, but this cannot be detected by normal debugging tools and profilers that track metrics like CPU usage.

### 2.5.3 Source code management

I used Git for version control and regularly pushed my local changes to a private GitHub repository.

### 2.5.4 Ethical statement

The development of this project did not require human participants, so nobody was harmed during its implementation.

The software that has been developed contributes to an existing blockchain ecosystem. Blockchains have many positive applications, but by their nature, they can facilitate the creation of exploitative markets. Since this software is already widely available, this project will not further this exploitation. Additionally, permissioned blockchains alleviate some of these harms as they are deployed in more controlled environments, and do not require energy-intensive proof of work mechanisms.

[4]The report says 97.89% coverage, but the only uncovered code is the testing code itself.

Chapter 3

# Implementation

This chapter describes the architecture of my implementation (Section 3.1), concludes my theoretical explanation of HotStuff by discussing the chained algorithm and the pacemaker (Section 3.2), presents a full specification for HotStuff with a proof of correctness and liveness (Section 3.3), describes key optimisations implemented (Section 3.4), presents the load generator and experiment scripts which will be used in evaluation (Section 3.5), and give an overview of the repository structure (Section 3.6).

## 3.1 System Architecture

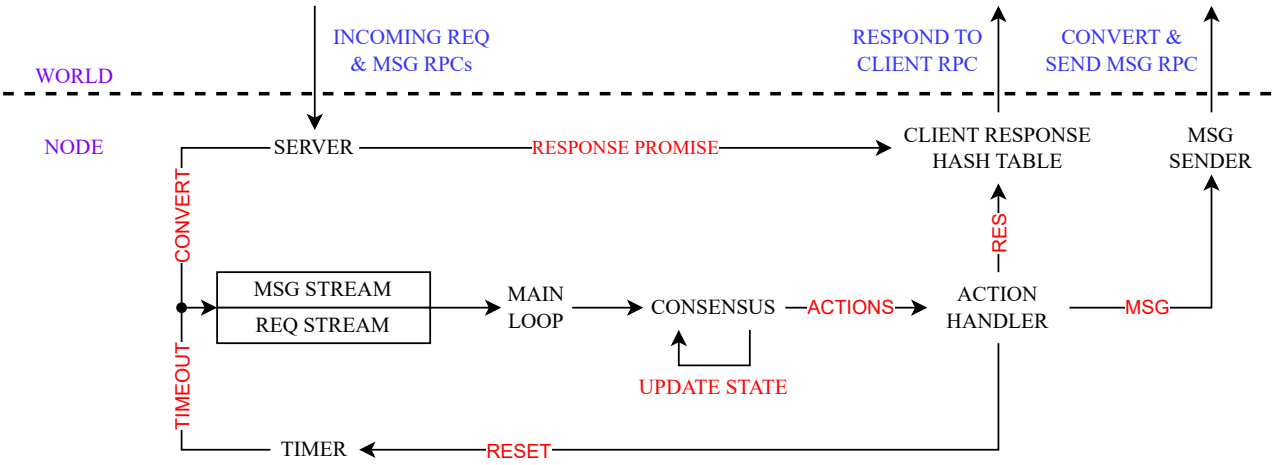



Figure 3.1: Architecture of a node.

This section presents the system architecture that surrounds the core *consensus* module, passing it new messages and client requests, and allowing it to send messages and respond to the client. This architecture is inspired by the OCons project[1] [37], which was developed by my project supervisor.

We will follow the path of an incoming request or message travelling through the system, as

[1]At the time I began implementation the OCons project was still under development, so I was unable to use the code in my project.

shown in Figure 3.1.

*Incoming message and client RPCs* — the node responds to internal messages from other nodes, and requests from a client (or a load generator, as described in Section 3.5.1) containing new commands to be committed. The format of these RPCs is specified in a Cap'n Proto schema, in their custom markdown language.

*Server* — each node operates as a server waiting for incoming RPCs. When the server receives an RPC it must be converted from Cap'n Proto types to internal types, and added to the *message stream* or *request stream*. If the RPC was a client request, a promise for a response is added to the *client response hash table*; this will allow the system to respond to the client once the command is decided.

*Message and request streams*[2] - messages and requests are added to separate streams so that messages can be prioritised. Internal messages represent a backlog of work that the system has not yet completed, so we follow the general design principle of clearing this backlog before accepting new work (client requests).

*Main loop* — takes messages and requests from their respective streams, and delivers them to the *consensus* module. This main loop ensures that the *consensus* module is never run in parallel, which could lead to race conditions.

*Consensus module* — takes incoming messages and requests, outputs *actions* such as sending messages, and updates its own state. The module contains an implementation of the basic HotStuff algorithm (Section 2.2), and the chained algorithm (chaining is discussed in Section 3.2.1, and a full specification is given in Section 3.3); both share the same signature, so can be interchanged. The module uses the Tezos cryptography library (Section 2.3.4) for signing messages, aggregating signatures, and checking quorum certificates.

*Action handler* — takes the actions outputted by the *consensus* module, and passes them to the appropriate handler. The three types of actions are: sending a message, responding to the client, and resetting the view timer.

*Message sender* — asynchronously dispatches an RPC in a new thread. To do this it must convert internal types into Cap'n Proto types, and construct an RPC that matches the schema. The message sender maintains TCP connections with all other nodes, and in the event of the connection breaking repeatedly attempts to reconnect with binary exponential back-off times.

*Client request hashtable* — allows client requests to be responded to. The hashtable maps each command's unique identifier to a promise, that will be awoken to respond to the original client request RPC once the command is committed.

*View timer* — waits for a timeout to elapse then adds a view timeout message to the *message stream* so that it will be delivered to the *consensus* module. The *reset* action allows the timer to be reset for a new view.

[2] A stream is a thread-safe implementation of a queue in Lwt.

## 3.2 More HotStuff theory

This section concludes my theoretical explanation of HotStuff by discussing chaining, and the pacemaker. As the pacemaker was not sufficiently specified in the original paper, I will draw on other sources to give a full explanation.

### 3.2.1 Chaining

This section describes the chained HotStuff algorithm, which is an optimised version of the basic algorithm described in Section 2.2 where different phases are pipelined. This is a standard optimisation for consensus algorithms that is described in the original paper [1].

Pipelining phases both simplifies and optimises the basic algorithm. The phases in the basic algorithm were all very similar; they involved collecting votes from replicas to form a QC that then serves in later phases. Each of the four phases[3] can be carried out concurrently. In each view, a leader collects votes to form a QC which can serve in all of the concurrent phases, then sends this QC to the next leader in a *new-view* message and broadcasts a proposal to the nodes.

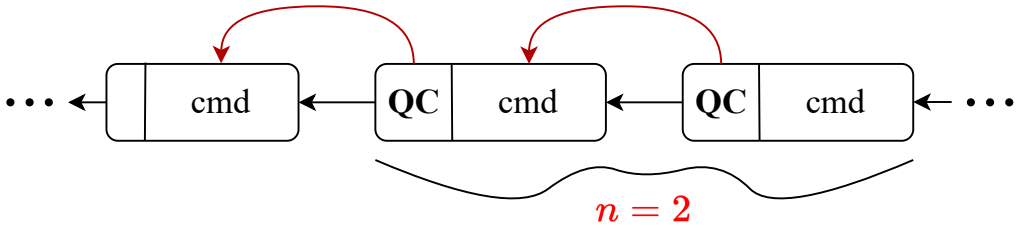



Figure 3.2: Sequence of nodes forming a 2-chain.

In each proposal, a chain of nodes (equivalent to the log) is extended, and different phases are carried out on suffixes of the chain depending on their length. For example, Figure 3.2 shows a 2-chain, which means a proposal has already been through 2 phases; this is the equivalent of being in the *commit* phase in the basic algorithm.

### 3.2.2 Pacemaker

This section discusses the pacemaker, which synchronises the views of honest replicas: a necessary condition for liveness (Property 2.2.2). The original paper did not specify the pacemaker mechanism, so I have synthesised information from different sources (including experimentation) to give a full specification of HotStuff with a pacemaker in Section 3.3. Part of the pacemaker that will be explained in this section is the view-change protocol, which allows the view of a faulty leader to be skipped; ours is based on the pacemaker for LibraBFT [9, 38]. The discussion will also cover the integration of the pacemaker with the

[3]Excluding the *new-view* phase which is unchanged.

HotStuff algorithm, which was achieved through modifications aimed at resolving deadlocks encountered during implementation.

There are two properties that a pacemaker must possess to provide the conditions needed for liveness [39]; it will later be proven that my pacemaker has these properties (Section 3.3.2). The first property, *view synchronisation* (Theorem 3.3.1), ensures that there are an infinite number of views with an honest leader that the non-faulty nodes stay in long enough to make progress. The second property, *synchronisation validity* (Theorem 3.3.4), ensures that the pacemaker will only advance to the next view if some honest replica wishes it to happen.

Pacemakers are related to failure detectors: mechanisms that facilitate the detection of failed nodes [40, 41]. A pacemaker extends this idea, allowing it to be used in a multi-shot setting.

### View-change protocol

Once the view times out, nodes send a *complain* message to the next leader and start a new timeout for the next view. Once the next leader achieves a quorum of *complain* messages it collects them into a QC known as a *view-change proof*. This leader can then send a *view-change* message containing the *view-change proof* to all replicas, who will respond by transitioning to the next view and sending a *new-view* message to the new leader. The inclusion of the *view-change proof* prevents liveness attacks by byzantine nodes that could otherwise attack the system by constantly causing view-changes to take place and preventing non-faulty leaders from making progress.

Crucially this protocol maintains the linear view change property that HotStuff has $O(n)$ authenticator complexity. Authenticator complexity measures the total number of threshold signatures and partial signatures in a view. This protocol requires $n - f$ partial signatures for the *complain* messages, and a single threshold signature for the *view-change* message, resulting in $O(n)$ authenticators overall.

### Integrating the pacemaker with HotStuff

My approach to integrating the pacemaker is to advance a node to the next view as soon as possible once it has finished proposing or voting, or when it receives evidence that there is a quorum of nodes in a higher view. This means that the system does not require synchronised clocks; the nodes advance asynchronously as fast as the network allows.

During development, I experimented with a prototype for a pacemaker that advanced views at a steady rate, but analysis of timing data indicated that this approach had inferior performance to my chosen design.

## 3.3 Specification

In this section, a full specification of HotStuff based on the basic HotStuff algorithm (Section 2.2), integrating chaining (Section 3.2.1), and a pacemaker (Section 3.2.2) is given. The changes made from the original algorithm are informally justified, and a proof of correctness for the modified algorithm is presented.

The consensus module is implemented according to the system architecture described in Section 3.1. Messages and requests are received from the main loop, and actions (sending a message, responding to a client request, or resetting the view timer) are outputted while the module updates its own state.

The pseudocode is presented with additions coloured in green and modifications in pink. Only the main changes are shown, excluding other features and performance improvements made (such as batching), which are presented in Section 3.4. The format of the event-driven pseudocode from the original paper is followed to allow for easier comparison and ease of implementation.

### 3.3.1 Changes to the original algorithm

This section informally justifies some of the key changes made to the original algorithm, with a specific focus on the changes I made to integrate the pacemaker with HotStuff (Section 3.2.2). Many of these changes were made to fix deadlocks encountered during implementation.

*Algorithm 1, line 28* — if a replica is the next leader, it waits to receive votes before transitioning to the next view. This prevents a deadlock where the leader transitions to the next view too early and ignores vote messages from an earlier view.

*Algorithm 1, line 31* — a node collects vote messages from future views so that if it has fallen behind, it can receive a quorum of votes and catch up to the current view. This prevents an honest node from falling behind and not being able to make progress in the view where it is the leader. Votes from different future views are stored in separate sets ($V_{\text{m.view}}$), to prevent votes from different views being used to form a QC.

*Algorithm 1, line 39* — proposals now include a QC, allowing replicas to catch up if they are in a lower view.

*Algorithm 2, line 1* — I have chosen to use a round-robin system to assign leaders to views.

*Algorithm 2, line 10* — as soon as a node transitions into a view where it is leader, ONBEAT is invoked, causing it to propose a new value.

*Algorithm 2, line 30* — if a node receives any QC from a future view $v$, it can safely transition to view $v + 1$

**Algorithm 1** Modified HotStuff

1: **function** CREATELEAF($parent,\ cmd,\ qc$)
2: $b.parent \leftarrow$ branch extending with dummy nodes from $parent$ to height $curView$
3: $b.height \leftarrow curView + 1$
4: $b.cmd \leftarrow cmd$
5: $b.justify \leftarrow qc$
6: **return** b
7: **procedure** UPDATE($b^*$)
8: $b" \leftarrow b^*.justify.node$
9: $b' \leftarrow b".justify.node$
10: $b \leftarrow b^*.justify.node$
11: UPDATEQCHIGH($b^*.justify$)
12: **if** $b'.height > b_{lock}.height$ **then**
13: $b_{lock} \leftarrow b'$
14: **if** $(b".parent = b') \wedge (b'.parent = b)$ **then**
15: ONCOMMIT($b$)
16: $b_{exec} \leftarrow b$
17: **procedure** ONCOMMIT($b$)
18: **if** $b_{exec}.height < b.height$ **then**
19: ONCOMMIT($b.parent$)
20: EXECUTE($b.cmd$)
21: **procedure** ONRECEIVEPROPOSAL($\text{MSG}_v(\text{GENERIC}, b_{new}, qc)$)
22: **if** $v = \text{GETLEADER}(m.view) \wedge m.view = curView$ **then**
23: $n \leftarrow b_{new}.justify.node$
24: **if** $b_{new}.height > vheight \wedge (b_{new}$ extends $b_{lock} \vee n.height > b_{lock}.height)$ **then**
25: $vheight \leftarrow b_{new}.height$
26: SEND(GETLEADER(), $\text{VOTEMSG}_u(\text{GENERIC}, b_{new}, \perp)$)
27: UPDATE($b_{new}$)
28: **if** not ISNEXTLEADER() **then**
29: ONNEXTSYNCVIEW($curview + 1$)
30: **procedure** ONRECEIVEVOTE($\text{VOTEMSG}_v(\text{GENERICACK}, b, \perp)$)
31: **if** $\text{ISLEADER}(m.view + 1) \wedge m.view \geq curView$ **then**
32: **if** $\exists(v, \sigma') \in V_{\text{m.view}}[b]$ **then**
33: **return**
34: $V[b] \leftarrow V_{\text{m.view}}[b] \cup \{(v, m.partialSig)\}$
35: **if** $|V_{\text{m.view}}[b]| \geq n - f$ **then**
36: $qc \leftarrow QC(\{\sigma | (v', \sigma) \in V_{\text{m.view}}[b]\})$
37: UPDATEQCHIGH($qc$)
38: ONNEXTSYNCVIEW($m.view + 1$)
39: **function** ONPROPOSE($b_{leaf}, cmd, qc_{high}$)
40: $b_{\text{new}} \leftarrow \text{CREATELEAF}(b_{leaf},\ cmd,\ qc_{high},\ b_{leaf}.height\ +\ 1)$
41: BROADCAST($\text{MSG}_v(\text{GENERIC}, b_{new}, qc_{high})$)
42: **return** $b_{\text{new}}$

**Algorithm 2** Modified Pacemaker

```
1: function GETLEADER
2:     return curView mod nodeCount
3: procedure UPDATEQCHIGH(qc'_high)
4:     if qc'_high.node.height > qc_high then
5:         qc'_high ← qc_high
6:         b_leaf ← qc'_high.node
7: procedure ONBEAT(cmd)
8:     if u = GETLEADER() then
9:         b_leaf ← ONPROPOSE(b_leaf, cmd, qc_high)
10: procedure ONNEXTSYNCVIEW(view)
11:     curView ← view
12:     RESETTIMER(curView)
13:     ONBEAT(cmds.take())
14:     SEND(GETLEADER(), MSG_u(NEWVIEW, ⊥, qc_high))
15: procedure ONRECEIVENEWVIEW(MSG_u(NEWVIEW, ⊥, qc'_high))
16:     UPDATEQCHIGH(qc'_high)
17: procedure ONRECIEVECLIENTREQUEST(REQ(cmd))
18:     cmds.add(cmd)
19: procedure ONTIMEOUT(view)
20:     SEND(GETNEXTLEADER(), MSG(COMPLAIN, ⊥, ⊥))
21:     RESETTIMER(view + 1)
22: procedure ONRECIEVECOMPLAIN(m = MSG(COMPLAIN, ⊥, ⊥))
23:     if ISLEADER(m.view + 1) ∧ m.view ≥ curView then
24:         if ∃(v, σ') ∈ C_m.view[b] then
25:             return
26:         C_m.view[b] ← C[b] ∪ {(v, m.partialSig)}
27:         if |C_m.view[b]| = n − f then
28:             qc ← QC({σ|(v', σ) ∈ C_m.view[b]})
29:             BROADCAST(MSG(NEXTVIEW, ⊥, qc))
30: procedure ONRECEIVEANY(m = MSG(∗, ∗, qc))
31:     if qc.view ≥ curView then
32:         ONNEXTSYNCVIEW(qc.view + 1)
```

### 3.3.2 Proofs

In this section, the correctness of the specification is proven. The safety (Property 2.2.1) of the specification holds trivially, as the changes that have been made do not invalidate the argument for the safety of the generic Byzantine algorithm (Section 2.2.2). Similarly, liveness (Property 2.2.2) still holds from the previous argument (Section 2.2.3). The only thing that remains to be proven is that the pacemaker has the properties that are needed to provide the conditions for liveness, as discussed in Section 3.2.2. These properties ensure that all honest replicas will remain in some view with an honest leader for long enough to make progress.

For this proof, the consensus machine (algorithm 1) and the pacemaker (algorithm 2) are considered to be separate entities. Furthermore, the assumption is made that GST has been

reached and messages have a bounded latency of $\Delta$ (assumption 2.2.1).

**Theorem 3.3.1** (View synchronisation)**.** *There exists infinite views $v_k$ and time intervals $\mathcal{I}_k$ such that the following holds:*

1. *The leader of $v_k$ is honest.*
2. *All honest replicas are in view $v_k$ for the duration of $\mathcal{I}_k$.*
3. *$\mathcal{I}_k$ is long enough for the replicas to make progress*

**Lemma 3.3.2.** *There exists an infinite number of consecutive assignments of two honest leaders to views. That is, we can always find future consecutive views $v_1$ and $v_2$ with honest leaders.*

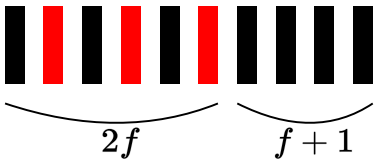



Figure 3.3: Example of round-robin leader allocation for $f = 3$. Red rectangles denote byzantine leaders.

*Proof.* A round-robin system allocates leaders to views. If we attempt to alternate honest and byzantine leaders, there will always be $f + 1$ consecutive honest leaders left over (Figure 3.3). Hence there will always be at least 2 consecutive honest leaders (the lemma holds trivially for $f = 0$). □

**Lemma 3.3.3.** *Every honest replica $x$ will eventually become the leader of a view.*

*Proof.* In the event that a byzantine leader tries to prevent honest nodes from transitioning to a higher view, the honest nodes will eventually timeout and send a COMPLAIN message to the next leader (algorithm 2, line 20). This may repeat if the next leader is also byzantine. Eventually, the COMPLAINs will be sent to an honest leader, that will send a NEXTVIEW message and transition all replicas into a new view (algorithm 2, line 29). Since $x$ will always progress to a higher view, it will eventually reach a view where it is the leader. □

*Proof of Theorem 3.3.1.* From lemma 3.3.2 we have that we can always find future views $v_1$ and $v_2$ with honest leaders $l_1$ and $l_2$, and from lemma 3.3.3 we have that $l_1$ will eventually enter $v_1$. We argue that all honest replicas will simultaneously be in either $v_1$ or $v_2$. Consider the cases of how $l_1$ could have entered $v_1$:

1. $l_1$ received a quorum of votes (algorithm 1, line 38) — $l_1$ will broadcast a QC of votes that will be received by all honest replicas within $\Delta$. These replicas will transition to $v_2$, and send a vote to $l_2$ which will also transition once it receives a quorum of votes. Hence all honest replicas will simultaneously be in $v_2$.

2. $l_1$ receives QC of COMPLAINS from itself (algorithm 1, line 32) — $l_1$ must have broadcast the NEXTVIEW message to all honest replicas; they will receive it within $\Delta$ and all enter $v_1$ simultaneously.

From this, we have that there is some time interval $\mathcal{I}_k$ throughout which all honest replicas are in view $v_k$, which has an honest leader. Once the replicas have entered $v_k$ they will only transition to the next view once they have made progress, or once they timeout, which will not happen assuming the timeout is sufficiently long. Hence $\mathcal{I}_k$ is long enough for the replicas to make progress. □

**Theorem 3.3.4** (Synchronisation validity)**.** *The pacemaker advances the view only if at least one honest consensus machine requests it to be advanced.*

*Proof.* This holds trivially for the calls to ONNEXTSYNCVIEW in algorithm 1, as the view is advanced on the request of the consensus machine.

The only other way the view can be advanced is on the receipt of a QC (algorithm 2, line 32). For a QC to be formed a quorum of $n - f$ nodes must have complained or voted; at least one of these must have been an honest consensus machine that requested for the view to be advanced. □

## 3.4 Practical challenges and optimisations

This section presents solutions to some of the practical challenges of implementing HotStuff, and describe optimisations I made to improve performance. This will cover how I implemented batching that was effective in increasing goodput, and the difficulties of designing a type for node chains and doing so efficiently. The debugging and experimentation that led to these solutions was non-trivial; they involved extensive analysis of the program trace and timing data, and the development of different prototypes to compare performance.

One main practical consideration was designing the types and structure of the core consensus module (Section 3.1), to implement the algorithm given in the specification (Section 3.3). Recall that this module takes as input messages and client requests (*events*), and returns actions (such as sending a message) and an updated consensus state. I used *variants* to tag the different types of events, with attached *records* to store the body of the event (such as the node and QC of a proposal). The core algorithm is expressed as a *match* statement to handle each type of event. The output from this is a list of actions (also implemented as variants) and a new state.

### 3.4.1 Batching

This section discusses the practical challenges of implementing effective *batching*. Batching is a standard technique to improve the goodput of a consensus algorithm by making each node

contain many commands instead of just one. This allows a single view to result in many commands being committed rather than just one.

A naive implementation of batching is simple to implement; instead of taking a single command from the queue to propose (algorithm 2, line 13), the whole queue can be batched into a single proposal. Analysis of the timing data for the naive implementation showed that it dramatically increased latency.

The rest of this section describes further optimisations I made to make batching effective.

### Filtering commands from batches

One optimisation that I implemented to make batching effective is filtering incoming commands to prevent them from being proposed by multiple nodes. This is achieved by nodes maintaining a set of commands that they have seen in the proposals of other nodes, and filtering these commands from their proposal. This optimisation increased the effectiveness of batching, which will be demonstrated in an ablation study (Section 4.3.3).

**Algorithm 3** Filtering implementation

```
1:  procedure ONRECEIVEPROPOSAL(MSG_v(GENERIC, b_new, qc))
2:      if v = GETLEADER(m.view) ∧ m.view = curView then
3:          seen ← seen ∪ b_new.cmds
4:          // ...
5:  procedure ONNEXTSYNCVIEW(view)
6:      curView ← view
7:      ONBEAT(cmds \ seen)
8:      cmds = seen = {}
9:      // ...
10: procedure ONRECIEVECLIENTREQUEST(REQ(cmd))
11:     cmds ← cmds ∪ {cmd}
```

Pseudocode for an implementation of filtering is given in algorithm 3. A set is used to store both the commands that are waiting to be proposed, and commands that have been seen, so that the difference can be efficiently computed. Also note the small optimisation on line 8: the seen set can safely be emptied as the commands have already been filtered, reducing the amount of computation required to calculate the set difference next time.

This optimisation is effective as the load generator is send-to-all (Section 3.5.1), so a new command is added to the queue of all nodes, and may be included in the proposals of different nodes. Filtering out commands so that they are not proposed multiple times follows the general design principle:

**Design principle:** Attempt to minimise the amount of redundant work that the system carries out by screening incoming work to check that it needs to be done.

An alternative implementation of filtering that was prototyped was maintaining a set of commands that had been committed rather than seen, but this filtered out far fewer commands

and did not significantly improve performance. This is likely because it takes several views for a command to be committed, and in this time multiple nodes may propose the same command and it will not be filtered.

#### Batch sizes

I further improved the effectiveness of batching by limiting the size of each batch. I demonstrate that this approach improves performance in a study of the system with different batch size limits (Section 4.3.1).

Implementing this feature requires minimal changes to algorithm 3, one simply has to take a subset of *cmds* to propose instead of the whole set. It is important to take the oldest commands from *cmds* to include in the proposal, so that older commands are not starved by newer ones, resulting in high latency. This can be accomplished by using an ordered set (such as a tree set) and ordering by command identifiers, which are ascending integers in my implementation.

The optimisation is effective because at higher message sizes the latency of Cap'n Proto serialisation increases (Section 4.2.1), so smaller batches can lead to better performance. There is an inherent trade-off between increasing batch size to commit more commands, and messages becoming slower due to increased serialisation latency.

### 3.4.2 Node chains

This section concerns the challenges of designing a suitable type for node chains (nodes[4]) that can be efficiently serialised by Cap'n Proto and sent over the network. I will discuss the challenges of designing the node type that must be overcome to allow the system to operate, then describe further optimisations that can be made. Recall from the discussion of the system architecture (Section 3.1) that the internal node type must be converted into a Cap'n Proto type in order to be serialised.

My initial naive implementation of the node type was an OCaml record that contained the fields *parent*, *cmds*, *height*, and *justify*. The *justify* field contained a QC that contains another node inside it.

#### The problem with recursive types

One problem encountered in converting internal types to Cap'n Proto types was that some nodes are recursive; the node stored inside the *justify* field points to themselves. This is the case in the chained HotStuff algorithm (Section 3.2.1); it has a recursive genesis node $b_0$ that starts the chain of nodes. The genesis node $b_0$ contains a hardcoded link to itself, so $b_0.justify.node = b_0$.

[4]Node is another word for log in this context, not to be confused with a node in the system.

This recursion poses a problem when carrying out the conversion between Cap'n Proto types and OCaml types. It is perfectly possible to define a recursive type in OCaml, so one can represent $b_0$ inside the consensus state machine. However, the naive implementation of a function to convert this node into a Cap'n Proto type will not terminate, as it will infinitely recurse into the field $b_0$.*justify.node*.

A simple solution to this problem is to add a flag to the Cap'n Proto schema *is_* $b_0$; when this flag is enabled then the node is assumed to be equal to $b_0$. This prevents $b_0$ from ever having to be converted into a Cap'n Proto type or being sent over the network, it can instead be reconstructed as a recursive type in the consensus state machine of the receiver.

### Replacing the justify.node field with an offset

One problem of the naive node type implementation was that it caused the system to crash due to rapidly increasing memory usage; this can be prevented by replacing the *node* record inside the *node.justify.node* field with an integer offset into the chain. This dramatically decreased the memory usage of the function to convert from internal types to Cap'n Proto types. The inefficiency of this function was revealed through profiling the memory usage of the program using Memtrace (Section 2.5.2).

The source of this problem was the inefficient design of the *node.justify* field, which contained a whole node inside it. As shown in Figure 3.2, each node has two links to previous nodes in the chain through the *parent* field and the *node.justify.node* field. By having each of these fields contain a whole *node* record, much of the chain had multiple redundant copies, resulting in a very bloated *node* object that was very expensive to convert.

A solution to this problem is to store an integer offset to a node inside the *justify* field rather than a *node* record. This offset represents how many *parent* links away the node is, and so can be used to reconstruct all of the original information. To implement this another type *node_justify* was added, which is identical to *qc*, but with the field *node* replaced with *node_offset*. One must then convert between the *node_justify* and *qc* types to reconstruct the original data and follow the *node.justify.node* link.

### Optimising the node equality function

One optimisation of the node type that was implemented is storing a *digest* field in the node that is a hash over all of the other fields[5], enabling efficient equality checking of nodes. This means that two nodes can be compared by their digests without having to recurse through the entire chain; the digests being equal cryptographically guarantees that the whole chains are equal. The need for this optimisation was discovered by profiling the memory usage of the node equality function using Memtrace.

[5]Notably the hash can be computed over the *digest* field of the parent node rather than recursing through the whole chain.

#### Optimising by truncating nodes

In order to optimise the sending of nodes I implemented node truncation; this reduces the size of nodes being sent over the network by cutting off older parts of the chain that the receiver already knows about. This is effective because the size of messages being sent is a bottleneck in my implementation (Section 4.2.1). The effectiveness of this optimisation is demonstrated in an ablation study (Section 4.3.3).

In order to truncate the node, my implementation recurses into the node's *parent* field, then deletes the field at some chosen depth. The entire node can then be reconstructed at the receiver, by 'splicing' it back together with $b_{exec}$, which contains the node up to the point that has been executed. Splicing together the nodes is done by recursing into the truncated node until it is equal to $b_{exec}$, then setting the deleted parent field to $b_{exec}$. The node equality function will still work on truncated nodes because of my optimisation to use digests (Section 3.4.2); a node will still have the same digest once it is truncated.

One practical challenge of this approach is choosing a suitable depth such that there is enough information at the receiver to reconstruct the whole chain. If there is a gap between the truncated node and $b_{exec}$ at the receiver, this will lead to commands being missed out and not executed. This is a problem in the event that some node becomes isolated from the rest; it must be able to catch up to the others once the network partition is healed.

To overcome this problem I used a TCP-style approach. I included a field containing the height of the $b_{exec}$ node to the *propose*, *new-view*, and *complain* messages. Each node maintains a list of the $b_{exec}$ height of every other node. When making a proposal, the leader takes the minimum height from this list and truncates the node up to that depth. This ensures that every node that receives the proposal has enough information to reconstruct the entire log.

There are some cases when the leader does not receive the latest $b_{exec}$ of every other node before it makes a proposal. This means that the leader will not truncate the node as much as it could have. I optimised this by having a node send the entire list of all stored $b_{exec}$ heights rather than just its own, allowing the heights to propagate around the system more quickly.

## 3.5 Implementing for evaluation

This section describes the infrastructure that will be used to evaluate system performance in Chapter 4, including the scripting developed to automate running experiments.

### 3.5.1 Load generator

The load generator is responsible for sending client requests to the nodes of the system. One is able to vary the throughput that the load generator drives the system at, and the duration that it runs for before it sends *kill* messages to the nodes, ending the test. It is also

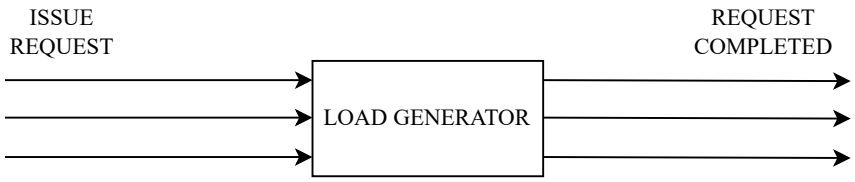



Figure 3.4: Open-loop load generator.

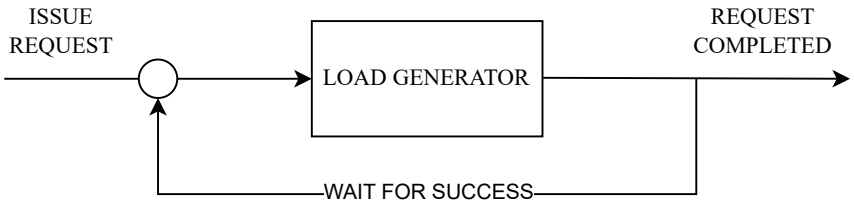



Figure 3.5: Closed-loop load generator.

responsible for timing and calculating statistics.

*Throughput* — the number of requests sent by the load generator each second.

*Goodput* — the number of requests that are responded to each second. This is calculated as the number of responses divided by the time difference between the first response and the end of the test.

*Latency* — the amount of time it takes between sending a request and receiving a response. The load generator reports the mean and standard deviation of latencies.

The load generator is open-loop (Figure 3.4), which means that it dispatches a request every $\delta$ seconds for the duration of the experiment, where $\delta = \frac{1}{throughput}$. This is in contrast to a closed-loop generator (Figure 3.5), which must wait until it receives a response in order to send the next request. An open-loop load generator is more useful as it allows us to overload the system and test its limits, whereas a closed-loop load generator waits for responses from the system, so cannot overload it.

The load generator is send-to-all, meaning that a command is sent to all nodes. An alternative is send-to-one, where a command is sent to a single node which is chosen at random. Send-to-all reduces latency as the next leader will have been sent the command, and may choose to propose it. In send-to-one it may take several views until the node that the request was sent to becomes the leader.

The load generator uses Lwt to asynchronously dispatch requests and stores a promise that will be fulfilled with their response. In the case of send-to-all the promises waiting on a response from each node are combined using `Lwt.pick`, meaning that the first node to respond will fulfil the promise and the rest will be ignored. Before beginning the experiment the load-generator sends 'dummy' requests to each node until all of them have sent a response; this ensures that all nodes are properly up and running before the experiment begins, reducing start-up effects.

### 3.5.2 Experiment scripts

Python scripts are used to automate the running of experiments. These scripts start the nodes and the load generator, wait for the experiment to run, kill the processes, run a script to plot graphs, then start the next experiment.

Different experiments may vary input variables such as throughput, batch size, and number of nodes. The script takes every permutation. Each experiment is repeated several times to reduce variance. Experiments are run in a random order so that if there is interference for some part of the test, this is not correlated with the parameters of the experiment, making anomalies easier to spot.

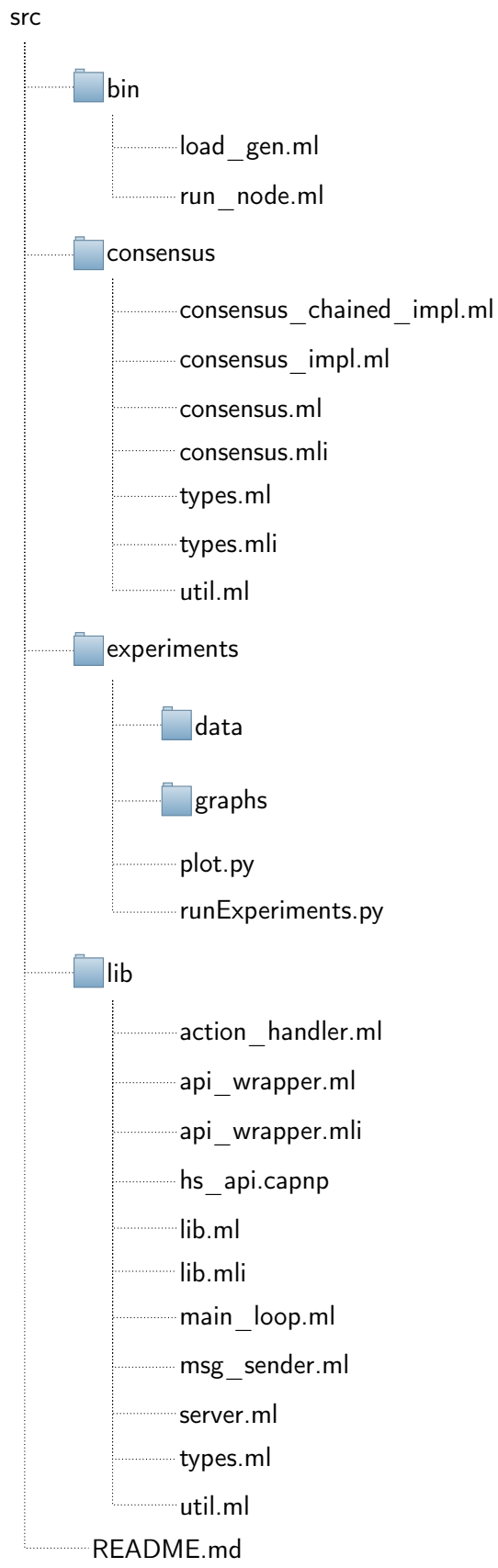
### 3.5.3 Logging framework

I developed a logging framework that stores the time taken for important parts of the program to execute and outputs key statistics such as the mean and standard deviation at the end of the test. This was essential for analysing the performance of the system to develop the optimisations described in Section 3.4.

The logging framework helped to reduce the effect of probing effects, where the behaviour of a system is altered by the act of measuring it. My previous approach printed the time taken throughout the execution of the program; since printing is CPU-heavy this resulted in probing effects. Notably, the print statements were not in-between the statements that measured the time taken, but they caused delays to happen in other parts of the program (presumably when the buffers were being flushed).

**Design principle:** Minimise probing effects by carrying out the minimum possible amount of work in critical areas of the program by storing data and moving work (such as outputting statistics) to less critical areas of the program.

## 3.6 Repository Overview




The *lib* directory contains files implementing the main components described in the system

architecture (Section 3.1). It also contains the schema for RPCs (*hs_ api.capnp*) and code to convert between internal types and Cap'n Proto types (*api_ wrapper.ml*).

Inside the *consensus* folder is a module with implementations of the basic and chained Hot-Stuff algorithms, sharing a common signature (*consensus.mli*). It also contains testing files to run expect tests (Section 2.5.2) that have been omitted for conciseness.

The *bin* folder contains the executables for running a node, and for running the load generator.

The *experiments* folder is where the data from running experiments is outputted to. It also contains scripts for running experiments and plotting graphs.

In order to run an experiment, first follow the instructions in *README.md* to set up your environment. You can then run experiments by executing `python3 runExperiments.py`, and modify this script to vary the input parameters such as throughput and experiment time. These scripts will run on Linux and MacOS, but will have to be modified to work on Windows.

Chapter 4

# Evaluation

This section highlights the methods and hardware used in evaluation (Section 4.1), benchmarks the performance of Cap'n Proto and the Tezos cryptography library (Section 4.2), and finally evaluates the performance of my HotStuff implementation (Section 4.3).

## 4.1 Testing methodology

The evaluation was carried out on the computer laboratory's Sofia server (2x Xeon Gold 6230R chips, 768GB RAM). Carrying out experiments on the server helped to minimise interference from other processes on the system.

Experiments were driven by an open-loop load generator (Section 3.5.1) and automated using Python scripts (Section 3.5.2). The load generator was run for varying amounts of time in the different experiments, with a further 15 seconds after this without the load generator running to wait for any slow responses.

To reduce the effect of interference, experiments were repeated three times, and the order of experiments was randomly permuted. Where a confidence interval is shown, this shows the range of results over the three repeats.

In experiments where throughput (in req/s) is varied, it starts at 1, increases exponentially (in multiples of 2) from 25 to 200 to benchmark performance at lower throughputs, and then increases linearly (by increments of 200) up to its maximum value (which varies between experiments).

## 4.2 Library benchmarks

This section presents benchmarks of the performance of the Cap'n Proto RPC framework, and the Tezos cryptography library.

### 4.2.1 Cap'n Proto

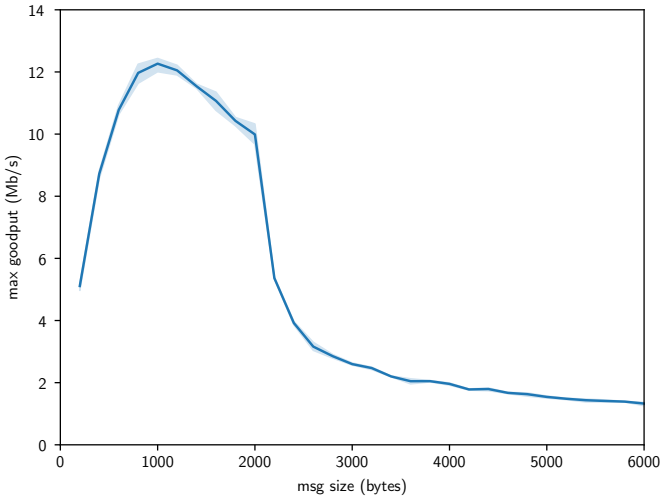



Figure 4.1: Benchmarking of Cap'n Proto server maximum send goodput for varying message sizes.

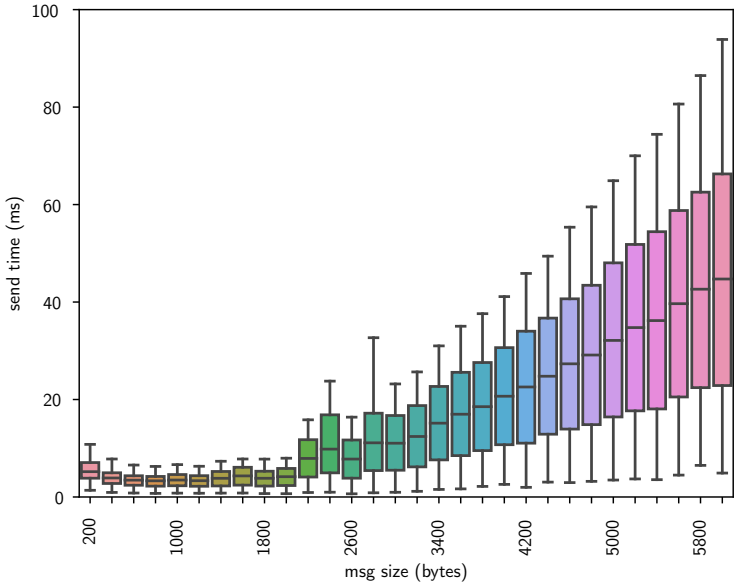



Figure 4.2: Benchmarking of Cap'n Proto send time for varying message sizes.

I benchmarked the performance of Cap'n Proto [32] for varying message sizes, measuring the maximum goodput at which messages could be sent (Figure 4.1), and the time taken to send (Figure 4.2). Message sizes (in bytes) were chosen to start at 1, increase exponentially (in multiples of 2) from 25 to 200 to benchmark performance for smaller message sizes, and then increase linearly (by increments of 200) up to 6000. Figure 4.2 is a box plot with whiskers plotted at the 5%ile and 95%ile with outliers excluded.

Cap'n Proto has an optimum message size of around 1000 bytes, at this point, data can be sent at the highest possible goodput (Figure 4.1). As the size of messages increases beyond this point, message serialisation costs cause the time taken to send messages to increase and

the maximum goodput that can be reached to decrease; there is a rapid increase in send time as messages increase beyond 2000 bytes (Figures 4.1 and 4.2)

### 4.2.2 Tezos Cryptography

I benchmarked key functions of the Tezos Cryptography library [33] with Jane Street's Core_bench module [42]. Core_bench is a micro-benchmarking library used to estimate the cost of operations in OCaml, it runs the operation many times and uses linear regression to try to reduce the effect of high variance between runs.

Cryptographic functions can take on the order of milliseconds to complete, with checking signatures demonstrated to be a particularly expensive operation (Table 4.1).

| Function | Time (µs) |
|---|---|
| Sign | 427.87 |
| Check | 1,171.77 |
| Aggregate (4 sigs) | 302.90 |
| Aggregate check (4 sigs) | 1,179.25 |
| Aggregate (8 sigs) | 605.38 |
| Aggregate check (8 sigs) | 1,180.61 |

Table 4.1: Benchmarking of key functions of the Tezos Cryptography library

## 4.3 HotStuff implementation benchmarks

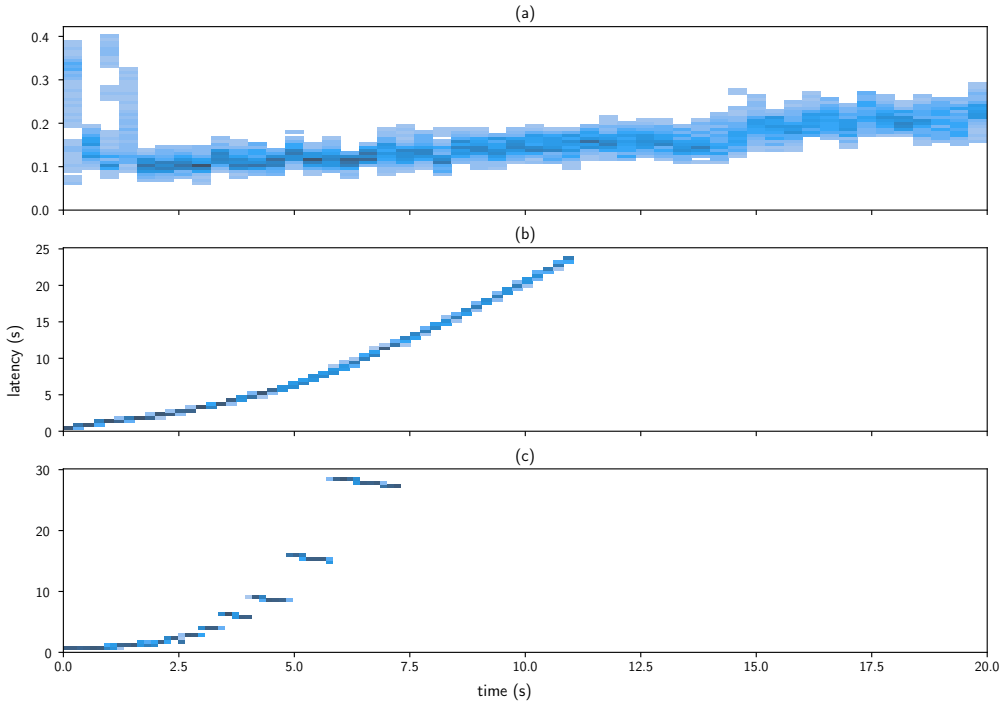



Figure 4.3: Heatmaps showing the behaviour of the system under varying conditions.

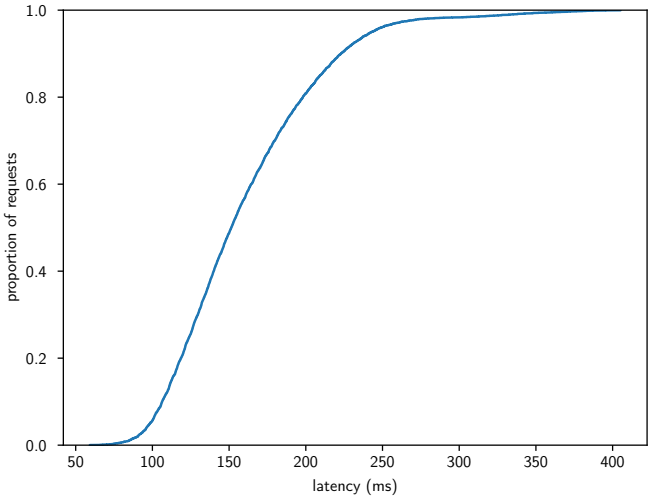



Figure 4.4: Cumulative latency plot for the system when exhibiting stable latency.

I now analyse the performance and behaviour of the system with different parameters and under different conditions. I argue that the optimisations described in Section 3.4 were effective in improving system performance, but there are fundamental limitations caused by the latency costs of Cap'n Proto serialisation (Section 4.2.1) and (to a lesser extent) cryptography (Section 4.2.2).

To illustrate the performance of the system under different conditions, several heatmaps (Figure 4.3) and a cumulative latency plot (Figure 4.4) are presented for tests run for 20s with 4 nodes.. Figure 4.3a and Figure 4.4 show an experiment with a throughput of 200req/s and a batch size of 300. Figure 4.3b shows an experiment with a throughout of 2000req/s and a batch size of 300. Figure 4.3c shows an experiment with a throughout of 2000req/s and unlimited batch sizes.

In most cases, the system exhibits stable latency throughout an experiment while goodput is equal to throughput, meaning that the system is not overloaded (Figure 4.3a, Figure 4.4). When the throughput exceeds the amount the system can keep up with, there is rapid growth in latency as commands queue on the nodes (Figure 4.3b). Since HotStuff is a partially synchronous protocol (Section 2.2), an increase in latency means that view times increase, decreasing goodput. Once the system is overloaded, the goodput levels off at around its maximum value as throughput is increased.

The comparison of batch sizes in Section 4.3.1 indicates that the batching implementation described in Section 3.4.1 is effective, as the system can achieve much greater goodput with batch sizes greater than 1 (equivalent to no batching). This section also provides evidence that serialisation latency is a bottleneck, as view times begin to increase exponentially as batch sizes increase (Figure 4.3c), due to messages being larger and taking longer to serialise.

The study of node counts (Section 4.3.2) gives further evidence that message serialisation is a bottleneck; higher node counts mean more internal messages being sent, causing a decline in performance due to serialisation costs. This also supports the conclusion that cryptography is a bottleneck, as more nodes mean more messages must be signed and aggregated.

The ablation study (Section 4.3.3) compares the performance of the system with different optimizations enabled, demonstrating their effectiveness in increasing goodput and lowering latency. It is also demonstrated that cryptography is a bottleneck, as there is an increase in latency with cryptography disabled.

In the wide area network (WAN) simulation study (Section 4.3.4), the performance of the system is evaluated in a simulated network (Section 2.5.2) with link latency similar to what may be observed in a wide area network.

In the view-change study (Section 3.2.2), it is shown that the view-change protocol (Section 3.2.2) effectively ensures the system progresses once a node has died, albeit with a significant performance penalty.

### 4.3.1 Batch sizes

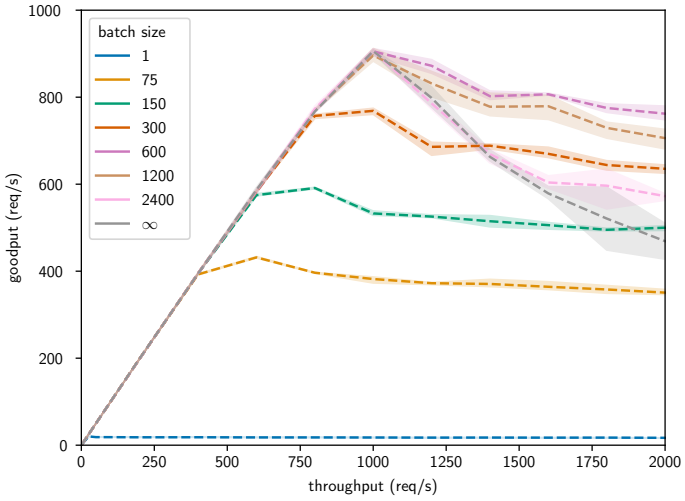



Figure 4.5: Benchmarking of goodput for varying throughputs and batch sizes.

This study compares the performance of the system for varying limits on batch sizes (Section 3.4.1). Experiments were run for 20 seconds on a network of 4 nodes. The experiment was run for longer to reduce the higher variance observed in tests where the system is overloaded. Figure 4.6 omits results with latency of above 1s, to show the performance of the system as it begins to be overloaded.

At lower throughputs the system is not overloaded; throughput grows linearly with goodput (Figure 4.5), as the system can respond to all incoming requests with roughly constant latency throughout an experiment (Figure 4.3a). During this period batches are not filled, so larger throughputs result in larger messages and a slow increase in latency due to increasing serialisation latency (Figure 4.6). The system can reach a higher goodput before being overloaded if it has a larger batch size, as each view results in more commands being committed; this supports the conclusion that batching is an effective optimisation.

Once throughput is increased enough, batches begin to be filled up and the system is overloaded. This results in the goodput flattening out (Figure 4.5), as the system cannot handle

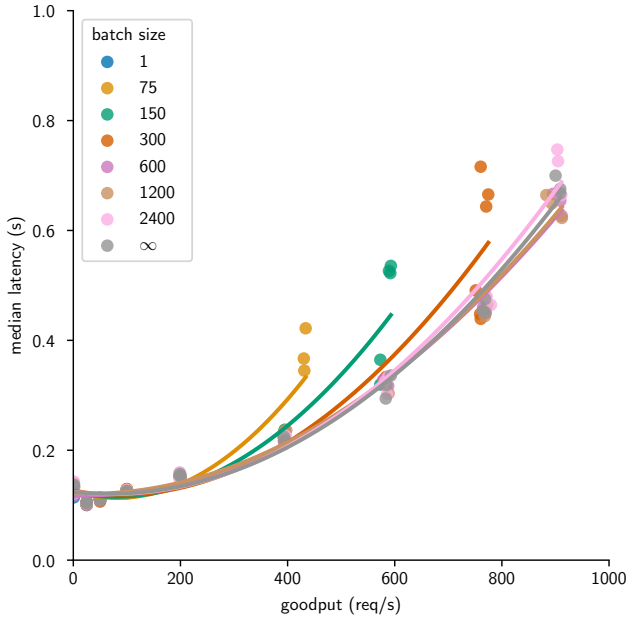



Figure 4.6: Benchmarking of goodput and median latency while varying throughputs and batch sizes.

the volume of requests; commands begin to queue on the nodes and latency rapidly increases throughout an experiment (Figure 4.3b, Figure 4.6). There is a slight decrease from the peak goodput due to the overheads of queueing.

For higher batch size limits (especially unlimited), the goodput declines more significantly once the system is overloaded. This is because the benefits of larger batches are offset by messages becoming larger, causing increased serialisation latency, which increases view times and lower goodput. For large batch sizes, view times increase exponentially, as shown by the growing vertical gaps between commands being committed in Figure 4.3c.

There is a clear trade-off between larger batch sizes that result in more commands being committed, and batches becoming too large and incurring exponential serialisation latency. The optimum for the system appears to be a batch size of around 600 commands, with a maximum goodput of around 900req/s (Figure 4.5).

### 4.3.2 Node counts

This study compares the performance of the system for varying node counts. Node counts were chosen such that $n = 3f + 1$ for some $f$, as choosing another value would decrease performance without any benefit of increased fault-tolerance[1]. All experiments were run for 10s with a batch size of 300. Figure 4.8 omits results with latency of above 1s, to show the performance of the system as it begins to be overloaded.

As node count increases the latency increases (Figure 4.8). This is because larger node counts mean that each view requires more internal messages to be sent to progress. Sending internal

[1]A node count of 2 was also tested as it is the smallest node count where internal messages are exchanged.

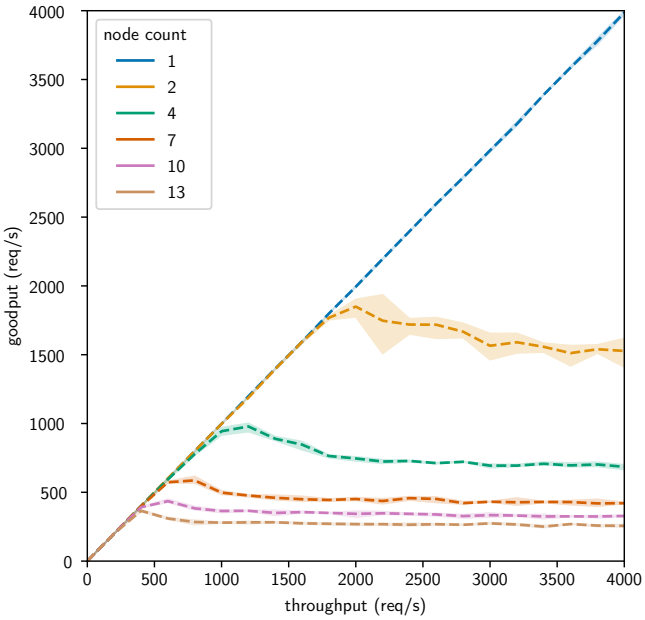



Figure 4.7: Benchmarking of goodput for varying throughputs and node counts.

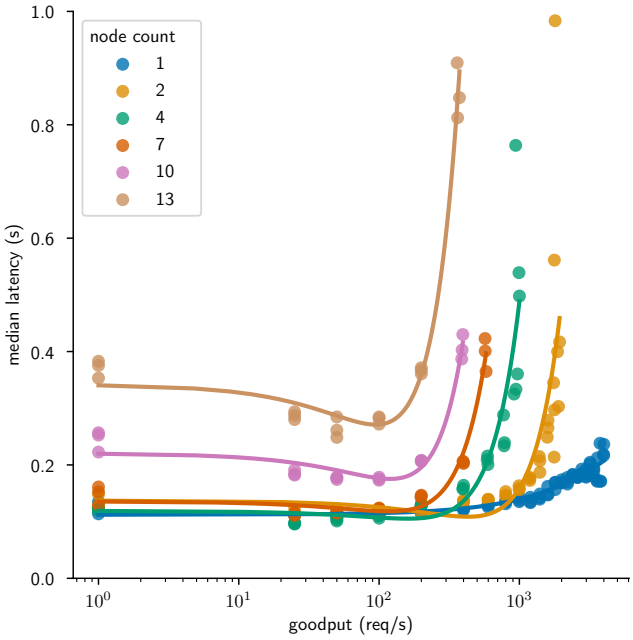



Figure 4.8: Benchmarking of goodput and median latency while varying throughputs and node counts.

messages is expensive due to the latency of serialisation and cryptography, so this results in increased overall latency. Additionally, increasing the node count increases the number of messages that must be signed, and makes aggregating signatures slower (Section 4.2.2).

Latency increases slowly with throughput while the system is not overloaded, then begins to rapidly increase once the system is overloaded and requests start to queue (Figure 4.8). Notably, the latency for a system of 1 node increases slowly, as there are no internal messages, just client requests and responses.

The larger the node count, the lower the maximum goodput that can be reached (Figure 4.7). This is again due to larger node counts resulting in more internal messages, causing more latency since this is a bottleneck. Increased latency causes each view to take longer, reducing the number of requests that can be responded to each second.

### 4.3.3 Ablation study

| Version | Chaining | Truncation | Filtering | Crypto |
|---|---|---|---|---|
| BASIC | ✗ | ✗ | ✗ | ✓ |
| CHAIN | ✓ | ✗ | ✗ | ✓ |
| FILT | ✓ | ✗ | ✓ | ✓ |
| TRUNC | ✓ | ✓ | ✗ | ✓ |
| ALL | ✓ | ✓ | ✓ | ✓ |
| NOCRY | ✓ | ✓ | ✓ | ✗ |

Table 4.2: Features enabled in different versions.

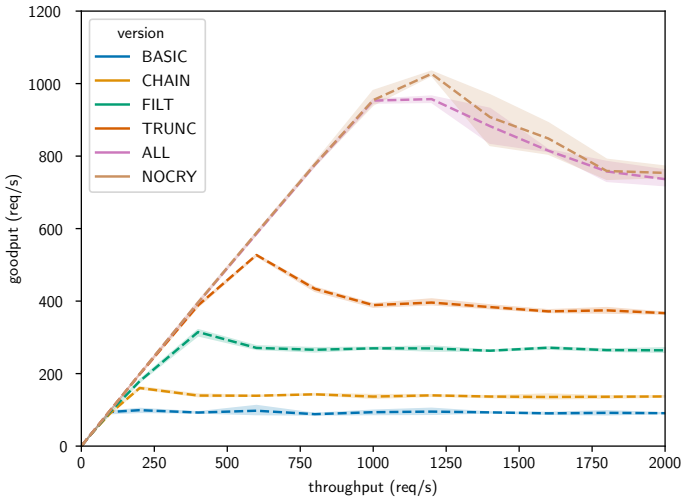



Figure 4.9: Benchmarking of goodput for varying throughputs and implementation versions.

This study compares the performance of the system with different optimisations enabled. The optimisations explored are chaining (Section 3.2.1), node truncation (Section 3.4.2), and command filtering (Section 3.4.1). Performance is also compared with, and without cryptography enabled. The mapping from version codes to which optimisations are enabled is given in table 4.2. The experiments displayed in Figure 4.9 and Figure 4.10 were run for 10s with a network of 4 nodes, and a batch size of 300. Figure 4.10 omits results with latency of above 1s, to show the performance of the system before it becomes overloaded.

The goodput and latency follow similar trends to Section 4.3.1 and Section 4.3.2.

The chained implementation is able to reach higher goodputs with slightly higher latency than the basic implementation. The higher goodput is a result of pipelining, more requests are able to transition through phases concurrently. However, pipelining also leads to the size of messages being increased, as each message contains batches of requests from each

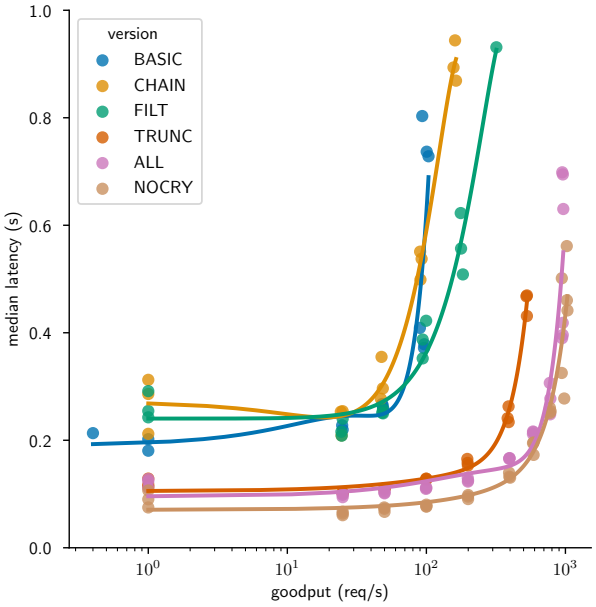



Figure 4.10: Benchmarking of goodput and median latency while varying throughputs and implementation versions.

concurrent phase; this leads to increased latency due to the time taken to serialise larger messages (Section 4.2.1).

Filtering significantly increases goodput, and slightly reduces latency. Goodput is increased as there are fewer redundant commands in each batch, so more useful commands can be processed. Latency is slightly reduced as messages are smaller on average due to some requests being filtered, so there is less latency due to serialisation costs.

Truncation dramatically reduces latency which leads to a large increase in goodput. This is because it significantly reduces the size of internal messages, reducing the latency incurred by serialisation costs.

Disabling cryptography reduces latency, leading to increased maximum goodput that can be reached. The reduction in latency is not that significant, implying that serialisation costs are more of a bottleneck than cryptography.

### 4.3.4 Wide area network simulation

I benchmarked the performance of the system in a simulated Mininet network [35, 36] with link latency of 100ms between each node. All experiments were run for 10s on a system with 4 nodes and a batch size of 300. Figure 4.12 omits results with latency of above 4s, to show the performance of the system as it becomes overloaded. Figure 4.4 is a cumulative latency plot of an experiment with a throughput of 200req/s.

The system has significantly higher latency in the simulator due to the five round trips which are required by the HotStuff protocol: this is an inherent bottleneck. We observe a median latency of around 1.3s (Figure 4.12) until the system becomes overloaded when it reaches

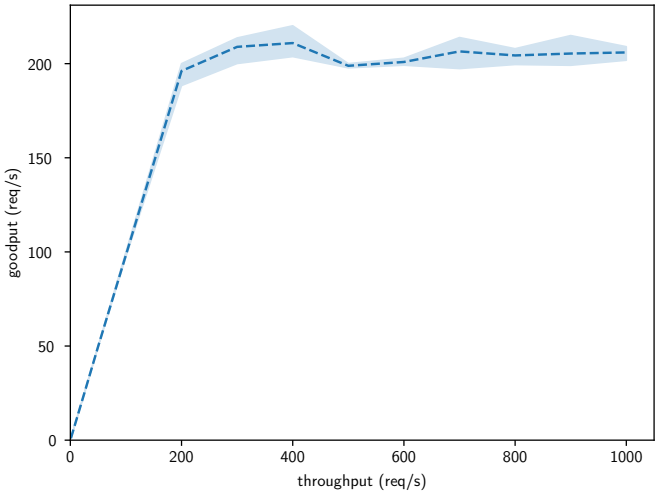



Figure 4.11: Benchmarking of goodput for varying throughputs.

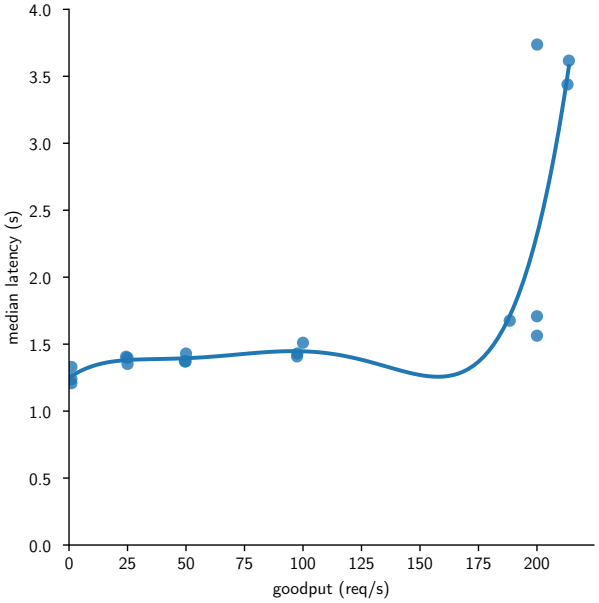



Figure 4.12: Benchmarking of goodput and median latency while varying throughputs.

a maximum goodput of 200req/s (Figure 4.11), and latency increases rapidly as commands queue on the nodes. One could predict a latency of around 1.1s in the simulated network, as a request takes 5 round trips to commit (1s total), and Figure 4.8 shows that the latency with negligible link latency was around 0.1s. The observed latency may be slightly greater (by 0.2s) as the nodes do not have synchronised clocks, so the increased link latency could cause delays in the view being advanced.

### 4.3.5 View-changes

This study explores the behaviour of the system in the event of a node failing, where the view-change protocol (Section 3.2.2) is needed to skip the faulty leader's view and make

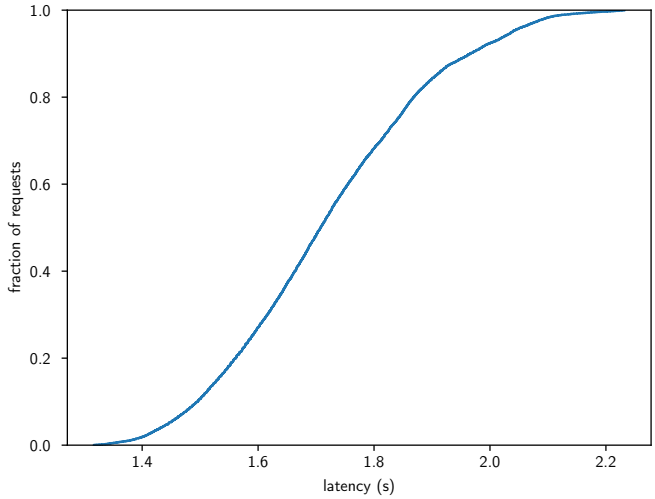



Figure 4.13: Cumulative latency plot for the system when exhibiting stable latency.

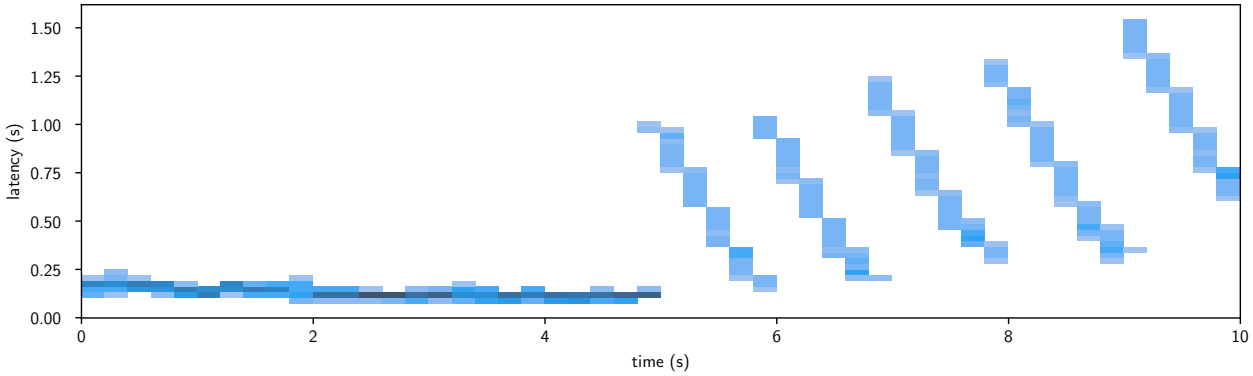



Figure 4.14: Heatmap of an experiment where a node is killed.

progress.

Figure 4.14 shows an experiment that was run for 10s on a network of 7 nodes with a batch size of 300. A node was killed 5s into the experiment. The view timeout was set to 0.5s.

There is a clear jump in latency every time the killed node is the leader of the view, and the nodes must wait for the 0.5s timeout to elapse before the next view begins. The latency jump of roughly 0.7s is about what one would expect; 0.5s timeout and 0.2s of latency (the same as before the node was killed). Latency gradually increases after this point as requests begin to queue on the nodes, incurring some overhead.

The view-change protocol is successful in allowing the system to make progress, albeit with a significant increase in latency. This is an inherent problem with the view-change protocol, although a failure-detector could help to detect that a node has failed and skip its view without waiting for a timeout to elapse, allowing the latency to return to a stable value.

# Chapter 5

# Conclusion

In this project I have given the first reference implementation of the HotStuff byzantine consensus algorithm in OCaml, contributing to the wider OCaml ecosystem[1]. The core algorithm is written in a self-contained module which could be reused by other projects with differing architectures and RPC systems. I have described some of the practical challenges of implementing HotStuff and implemented several optimisations of the basic algorithm (Section 3.4). One main challenge was adapting the HotStuff pacemaker; I gave a full specification and proved that the pacemaker has desirable properties (Section 3.2.2).

The project successfully meets the requirements set out in Section 2.4. There is significant evidence for the correctness of my implementation; the testing suite has 100% coverage of the consensus state machine code (Section 2.5.2). Evaluation of the system was carried out both locally and on a simulated network, and I analysed its behaviour with different parameters, and under varying conditions (Section 4). This analysis helped to identify that the system bottlenecks are message serialisation and cryptography. I also implemented several optimisations (Section 3.4), and demonstrated their effectiveness in an ablation study (Section 4.3.3).

Given that message serialisation and cryptography were shown to be system bottlenecks, future work could aim to overcome some of these problems to achieve better performance. One potential direction would be to implement custom message serialisation, and use a faster RPC library such as EIO [31]; both of these were out of the scope of this project. Alternative cryptography libraries could also be explored, although this may be an inherent bottleneck of any HotStuff implementation.

Future work could also explore the challenges of deploying a production-ready system based on my implementation. Although HotStuff is byzantine-fault tolerant, this project has not considered other security threats such as attacks on availability. Solving these problems would be non-trivial; some of my optimisations may be antagonistic with security considerations, for example, a malicious node could repeatedly request the whole chain to be sent (instead of truncated) and bring down the system. One could also implement a proof of work or proof of stake mechanism on top of my implementation to make it resilient to Sybil attacks, allowing it to be deployed in a permissionless setting.

One lesson I learnt from this project is that graphs are an invaluable tool for analysing the performance of a distributed algorithm, and analysis of graphs should be included in the

[1]Since the project is implemented purely in OCaml, it could be deployed in a MirageOS unikernel [43]

iterative passes of the waterfall development model (Section 2.5.1). Much of development time was spent debugging system performance to implement the optimisations described in Section 3.4. I found that as I was creating graphs and analysing performance (Section 4.3), I was able to more quickly find bugs and intuitively understand the behaviour of the system. Therefore if I were to do a similar project in future, I would further automate and parallelise the testing and graph plotting scripts to quickly gain insight during development.

In conclusion, this project has provided an implementation of a byzantine consensus algorithm, a key algorithm in the development of decentralised software. Decentralised software has far-reaching implications, and could challenge the control of large centralised authorities over critical infrastructure, platforms, and organisations.

# Bibliography


[1] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, Toronto ON Canada, July 2019. ACM. https://dl.acm.org/doi/10.1145/3293611.3331591.

[2] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Technical report, 2008. https://www.ussc.gov/sites/default/files/pdf/training/annual-national-training-seminar/2018/Emerging_Tech_Bitcoin_Crypto.pdf.

[3] Ingolf G. A. Pernice and Brett Scott. Cryptocurrency. *Internet Policy Review*, 10(2), May 2021. https://policyreview.info/glossary/cryptocurrency.

[4] Samer Hassan and Primavera De Filippi. Decentralized Autonomous Organization. *Internet Policy Review*, 10(2), April 2021. https://policyreview.info/glossary/DAO.

[5] Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. Technical report, 2014. https://ethereum.org/en/whitepaper/.

[6] Ethereum Name Service. https://ens.domains.

[7] Gavin Wood. Ethereum: A secure dentralised generalised transaction ledger. Technical report, 2022. https://ethereum.github.io/yellowpaper/paper.pdf.

[8] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982. https://dl.acm.org/doi/10.1145/357172.357176.

[9] M. Baudet, A. Ching, A. Chursin, G. Danezis, François Garillot, Zekun Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino. State Machine Replication in the Libra Blockchain. Technical report, 2019. https://www.semanticscholar.org/paper/State-Machine-Replication-in-the-Libra-Blockchain-Baudet-Ching/71bb5ddbc614c241200302da0a9a6130587e3793.

[10] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, USA, February 1999. USENIX Association. https://dl.acm.org/doi/10.5555/296806.296824.

[11] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580, June 2019. https://ieeexplore.ieee.org/document/8809541.

[12] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988. https://dl.acm.org/doi/10.1145/42282.42283.

[13] Jae Kwon. Tendermint : Consensus without Mining. 2014. https://www.semanticscholar.org/paper/Tendermint-%3A-Consensus-without-Mining-Kwon/df62a45f50aac8890453b6991ea115e996c1646e.

[14] Vitalik Buterin and Virgil Griffith. Casper the Friendly Finality Gadget. http://arxiv.org/abs/1710.09437, January 2019.

[15] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, August 2007. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/1281100.1281103.

[16] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014. https://www.usenix.org/node/184041.

[17] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980. https://dl.acm.org/doi/10.1145/322186.322188.

[18] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, March 1986. https://doi.org/10.1007/BF01843568.

[19] Charles H. Bennett, François Bessette, Gilles Brassard, Louis Salvail, and John Smolin. Experimental quantum cryptography. *Journal of Cryptology*, 5(1):3–28, January 1992. https://doi.org/10.1007/BF00191318.

[20] Ittai Abraham and Gilad Stern. Information Theoretic HotStuff. http://arxiv.org/abs/2009.12828, November 2020.

[21] Rafael Pass and Elaine Shi. Thunderella: Blockchains with Optimistic Instant Confirmation. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, Lecture Notes in Computer Science, pages 3–33, Cham, 2018. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-78375-8_1.

[22] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. https://dl.acm.org/doi/10.1145/279227.279229.

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

[24] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. https://dl.acm.org/doi/10.1145/359545.359563.

[25] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990. https://dl.acm.org/doi/10.1145/98163.98167.

[26] Victor Shoup. Practical Threshold Signatures. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and Bart Preneel, editors, *Advances in Cryptology — EUROCRYPT 2000*, volume 1807, pages 207–220. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. http://link.springer.com/10.1007/3-540-45539-6_15.

[27] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *Journal of Cryptology*, 18(3):219–246, July 2005. https://doi.org/10.1007/s00145-005-0318-0.

[28] OCaml Programming Language. https://ocaml.org.

[29] Ocsigen. Lwt Library. https://github.com/ocsigen/lwt.

[30] Jane Street. Async Library. https://opensource.janestreet.com/async/.

[31] MultiCore OCaml. Eio Library. https://github.com/ocaml-multicore/eio.

[32] Cap'n Proto RPC Library. https://capnproto.org/.

[33] Tezos Cryptography Library. https://opam.ocaml.org/packages/tezos-crypto/.

[34] Jane Street. Memtrace Memory Profiler & Viewer. https://opam.ocaml.org/packages/memtrace/.

[35] Mininet. http://mininet.org/.

[36] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 1–6, New York, NY, USA, October 2010. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/1868447.1868466.

[37] Chris Jensen. OCons Paxos Implementation. https://github.com/Cjen1/OCons.

[38] Ittai Abraham. Byzantine fault tolerance, state machine replication and blockchains. SPTDC, St. Petersburg, 2019. https://www.youtube.com/watch?v=Uoo2cOvQVN0.

[39] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems*, 1(2), October 2021. https://cryptoeconomicsystems.pubpub.org/pub/naor-cogsworth-synchronization/release/5.

[40] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996. https://dl.acm.org/doi/10.1145/234533.234549.

[41] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996. https://dl.acm.org/doi/10.1145/226643.226647.

[42] Jane Street. Core_bench. https://github.com/janestreet/core_bench.

[43] MirageOS. https://mirage.io/.

**Appendix A**

# Proposal

Marc Harvey-Hill

# Project proposal: Implementing the HotStuff consensus algorithm

Computer Science Tripos Part II

Gonville and Caius College

12 10 2022

## Description

This project aims to implement the HotStuff algorithm as outlined by Yin et al . in 2019 [1]. HotStuff is a Byzantine fault-tolerant algorithm that allows state to be replicated across nodes under a partially synchronous model. This has applications in blockchain, and can be used

as the foundation for the development of cryptocurrencies and decentralised applications.

A leader node will drive consensus, ensuring that non-faulty replicas run identical commands in the same order so that state is consistent across replicas. The algorithm has a two-phase process for reaching consensus on a given proposal - the first forms a quorum certificate to guarantee a unique proposal and the second guarantees that the next leader will be able to convince replicas to vote for a safe proposal. Additionally the algorithm has a three-phase process for a view-change, which involves selecting a new leader, allowing it to collect information, and make a proposal to replicas.

The project will be implemented in OCaml on top of existing RPC and cryptographic libraries such as AsyncRPC and tezos-crypto. It can be divided into three main sections:

- Core algorithm - a non-chained implementation without Byzantine fault-tolerance
- Cryptography - integration of cryptographic signing to ensure Byzantine fault-tolerance
- Chained - pipelined version of HotStuff allowing a quorum certificate to serve in multiple phases simultaneously.

## Starting point

I have some experience using OCaml from the IA course.

## Success Criteria

- Correctness - The consensus algorithm is implemented as it is described in the paper. This can be established by comparison of the program trace to a known correct implementation or mapping to a verified TLA+ model.
- Evaluation - Analysis of system throughput and latency carried out on a simulated network of 32 replicas.

Evaluation will be carried out by testing the program locally, analysing the trace, and testing in an emulator.

## Extensions

- Improve transaction throughput and reduce latency. This can be achieved through architectural decisions, tuning the scheduler, and ensuring cryptographic libraries are being used efficiently.
- Support reconfiguration of the network.

# Timetable

| Start | End | Work |
|---|---|---|
| 17/10/22 | 30/10/22 | Preparatory work:<br>Set up local environment and repository.<br>Learn OCaml and RPC library |
| 31/10/22 | 13/11/22 | Study HotStuff paper in depth.<br>Begin implementing core algorithm.<br>*Cloud Computing 1 deadline (9th Nov)* |
| 14/10/22 | 27/11/22 | Continue implementing core algorithm.<br>*Milestone: core algorithm implementation completed* |
| 28/11/22 | 11/12/22 | Integrate cryptographic libraries.<br>*Cloud Computing 2 deadline (28th Nov)* |
| 12/12/22 | 25/12/22 | Begin implementing chained algorithm. |
| 26/12/22 | 08/01/23 | *Christmas break* |
| 09/01/23 | 22/01/23 | Complete implementation of chained algorithm.<br>Test implementation locally.<br>*Milestone: success criteria met* |
| 23/01/23 | 05/02/23 | Write progress report, prepare presentation.<br>*Cybercrime 1 deadline (3rd Feb)*<br>*Milestone: Progress report submitted (3rd Feb)* |
| 06/02/23 | 19/02/23 | *Slack time / work on extensions*<br>*Cybercrime 2 deadline (17th Feb)*<br>*Milestone: Presentation delivered (8th - 15th Feb)* |
| 20/02/23 | 05/03/23 | Write dissertation outline.<br>*Cybercrime 3 deadline (3rd March)* |
| 06/03/23 | 19/03/23 | Write preparation and implementation chapters.<br>*Cybercrime 4 deadline (17th March)* |
| 20/03/23 | 02/04/23 | Write evaluation chapter. |
| 03/04/23 | 16/04/23 | Write introduction and conclusion chapters.<br>*Milestone: Dissertation draft completed and sent to supervisors and DoS* |
| 17/04/23 | 30/04/23 | Respond to feedback.<br>*Milestone: Dissertation completed* |
| 01/05/23 | 12/05/23 | *Slack time*<br>*Milestone: Dissertation submitted (12th May)* |

# Resources

Laptop (Macbook Air 2020 with M1 chip, 16GB RAM, 512GB SSD)
Sofia server (2x Xeon Gold 6230R chips, 768GB RAM)
git for version control and backups to Github. TeX for typesetting.
If there is a problem with my machine, I will clone the repository and continue on another one of my machines or the MCS.