Marc Harvey-Hill

# Implementing the HotStuff consensus algorithm

Computer Science Tripos Part II
Gonville and Caius College
March 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. I am content for my dissertation to be made available to the students and staff of the University.

Signed [signature]

Date [date]

# Proforma

blah blah

# Contents

**1 Introduction** **5**

**2 Preparation** **7**

2.1 Starting point . . . 7

2.2 HotStuff algorithm . . . 7

2.2.1 Non-byzantine consensus . . . 8

2.2.2 Byzantine consensus . . . 9

2.2.3 Optimistic responsiveness . . . 9

2.2.4 View-changes . . . 11

2.2.5 Chaining . . . 12

2.3 Tools & libraries . . . 13

2.3.1 OCaml . . . 13

2.3.2 Lwt . . . 13

2.3.3 Cap'n Proto . . . 14

2.3.4 Tezos cryptography . . . 14

2.4 Requirements analysis . . . 14

2.5 Software engineering practices . . . 15

2.5.1 Development methodology . . . 15

2.5.2 Testing & debugging methodology . . . 15

2.5.3 Source code management . . . 15

**3 Implementation** **16**

3.1 Architectural overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3.2 Pacemaker Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3.2.1 Proofs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

3.3 Performance improvements . . . . . . . . . . . . . . . . . . . . . . . . . . 21

3.3.1 Batching . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

3.3.2 Encoding of nodes . . . . . . . . . . . . . . . . . . . . . . . . . . 26

3.4 Implementing for evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 28

3.4.1 Load generator . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

3.4.2 Experiment scripts . . . . . . . . . . . . . . . . . . . . . . . . . . 29

3.5 Repository Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

**4 Evaluation** **32**

4.1 Testing methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

4.2 Library benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

4.2.1 Cap'n Proto . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

4.2.2 Tezos Cryptography . . . . . . . . . . . . . . . . . . . . . . . . . 34

4.3 HotStuff implementation benchmarks . . . . . . . . . . . . . . . . . . . . 34

4.3.1 Batch sizes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

4.3.2 Node counts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

4.3.3 Ablation study . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

4.3.4 Wide area network simulation . . . . . . . . . . . . . . . . . . . . 39

4.3.5 View-changes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

**5 Conclusion** **43**

**Bibliography** **45**

## Chapter 1

# Introduction

The power of blockchains lies in their ability 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, traditional corporations can be replaced with decentralised autonomous organisations (DAOs), internet infrastructure like DNS servers can be decentralised, and any possible algorithm can be run on a decentralised 'world computer'. The innovation that make blockchains possible is the byzantine consensus algorithm.

Byzantine fault-tolerant consensus algorithms allows 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 between many devices on a ledger of transactions like "Account X transfers account Y £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. In this problem, a group of generals must all agree to siege a castle at the same time, as a single army would be defeated on its own. The problem is that the generals 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 ensure that they all attack together. Some byzantine consensus algorithms are able to reach consensus on multiple values instead of just one, so instead of deciding a single value like "Attack at dawn", they can agree on a continuously growing log of multiple values. The key is that once a value is decided and appended to the log it can never be modified or erased, the log can only ever be extended. [*** maybe condense this paragraph]

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 and Ethereum, 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 permissioned blockchain can be overrun by a large number of malicious nodes; proof of work, for example, adds a requirement for a 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, a cancelled permissioned blockchain-based payments system. The algorithm is relevant because of various performance advantages over other byzantine consensus algorithms such as PBFT, DLS, Tendermint, and Casper.

Building practical, well-performing implementations of consensus algorithms is highly non-trivial. These algorithms are usually specified 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 [2].

The main contributions of this dissertation are:

- Providing a reference implementation of HotStuff in OCaml based on a paper by Yin et. al [3].
- Outlining key practical challenges and considerations of this implementation.
- We adapt the pacemaker mechanism presented in the HotStuff paper, giving a full specification that works in asynchronous environments without synchronised clocks. The paper did not sufficiently specify the pacemaker mechanism, which drives the system to ensure it makes progress.

Chapter 2

# Preparation

*In this chapter we disclose my knowledge and experience prior to beginning this project [2.1], give a theoretical basis for understanding the HotStuff algorithm [2.2], outline the tools, libraries [2.3], and methodology [2.5] deployed in implementation, and highlight the requirements that the implementation should meet [2.4].*

## 2.1 Starting point

I had some experience using OCaml from the IA course but had never used it in a project. I also had some background in distributed systems from the IB course, which briefly covered Raft, a non-byzantine consensus algorithm. I had some understanding of byzantine consensus from my own reading into Nakamoto consensus and from developing a wallet application for Ethereum; neither of these was directly useful to implementing HotStuff, but they gave me some wider context of the field.

## 2.2 HotStuff algorithm

A consensus algorithm allows a group of nodes to reach agreement 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 the *replicas* (another word for nodes). After several messages are exchanged the log may be committed, meaning that the log has been agreed upon up to that point and cannot be altered. Once the log is committed up to some point the leader can send 'decide' to the replicas; if the log is a list of commands, then the new commands can safely be executed. A consensus algorithm must have these properties:

- Safety: Once a log has been committed up to some point that part of the log is immutable, it can only appended to.
- Liveness: The system is guaranteed to make progress once a non-faulty leader is elected.

The adverse conditions that HotStuff can operate under are described by the system model: partially synchronous, byzantine, with reliable, authenticated, point-to-point delivery. This means that messages sent by one party will always be delivered to another within some bounded amount of time after global synchronisation time (GST) has been reached and a message source cannot be spoofed. The byzantine assumption means 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.

### 2.2.1 Non-byzantine consensus

We will start by describing an algorithm to solve the simpler problem of reaching consensus with the stronger assumption of a crash-stop model instead of a byzantine one; this means that we assume nodes cannot be malicious but they can still crash and never come back online. Examples of similar algorithms include Raft and multi-shot Paxos.

Each view can be broken into two phases; phase 1 allows the leader to learn of previously decided values, and in phase 2 it decides on a value. Phase 1 is initiated by the leader broadcasting the current view number to the replicas, which respond by sending their longest accepted log (the one with the highest corresponding view number). Once the leader has a quorum of responses, it initiates phase 2: it selects the longest log that has been sent to it and broadcasts it to the replicas. The leader may also extend the log at this point with its own values, or create a new log if it did not receive anything. Finally, each replica updates its log to the value sent by the leader and sends an acknowledgement. Once the leader receives a quorum of acknowledgements it can commit the new log. This algorithm satisfies our requirement that a committed log can only be extended.

We will refer to phase 1 as the *new-view* stage, and phase 2 as the *commit* phase to use the terminology of the HotStuff paper. These are followed by the *decide* phase, in which the leader sends a decide message to replicas. Once they receive this message they can consider the log decided, and execute the new commands which have been added to the log. See an example in figure 2.1.

1. `New-view:`
   - (a) `leader` → `replicas: “view = 3"`
   - (b) `replicas` → `leader: “view = 2, log = [‘hello’, ‘world’]", ...`
2. `Commit:`
   - (a) `leader` → `replicas: propose “log = [‘hello’, ‘world’, ‘!’]"`
   - (b) `replicas` → `leader: “ack", ...`
3. `Decide:`
   - (a) `leader` → `replicas: “decide"`
   - (b) `replicas: execute log`

Figure 2.1: Example of crash-stop consensus algorithm.

### 2.2.2 Byzantine consensus

In order to extend our algorithm to achieve consensus under a Byzantine threat model we must handle two threats that we will deal with in turn. To do this we must first introduce the concept of a 'threshold signature'. Acknowledgements from replicas all contain a signature over the message to prove that they were actually sent by the correct node. The leader can create a threshold signature by combining $n - f$ ack messages' signatures to prove that they really received a quorum of acknowledgements. A collection of a quorum of votes with a threshold signature is known as a 'quorum certificate' or 'QC'.

1. Threat: Equivocation — a faulty leader broadcasts one value to some replicas and a different value to others. For example in the case of a cryptocurrency, this could result in a malicious actor (controlling account X) carrying out a double spend attack, sending "Account X transfers account Y £10" to some nodes, and "Account X transfers account Z £10" to other nodes, even if Account X only contains £10.

   Solution: Add a new stage *prepare* which happens just before the *commit* phase. In this phase the leader again chooses the longest log it received in the *new-view* phase to pre-propose in the *prepare* phase; the log may also be extended with the leader's new values. Once the leader receives a quorum of acknowledgements, the *commit* phase begins. The difference is that this time the leader includes in its proposal a QC over *prepare* acks, which proves that it pre-proposed the value to at least $n - f$ nodes and received their acks, so it is not proposing one value to some nodes and another value to others.

2. Threat: A faulty leader sends in the *prepare* phase a log that conflicts with one that has already been committed.

   Solution: Replicas must *lock* on a value once it is committed and not accept a *prepare* from a leader that contradicts that. They will store the QC that they receive during the *commit* phase and will only accept a new *prepare* if it extends from the node stored in the certificate.

See an example in figure 2.2.

We give an informal inductive argument of the safety of this algorithm based on it solving these two threats. To be safe, we must have that if some log $\lambda$ is committed in view $v$, at no point in future will some conflicting log be committed. In view $v$ no log that conflicts with $\lambda$ may be committed, as we have prevented equivocation (threat 1). If we have in some view $v'$ that $\lambda$ (or some log extending it) is committed, then no conflicting log can be committed in view $v' + 1$. This holds because any log proposed in view $v' + 1$ must receive a quorum of $2f + 1$ votes in the *prepare* phase; there must be at least one honest node in the intersection between this quorum, and the quorum of $2f + 1$ nodes that voted to commit $\lambda$ in view $v$. This honest replica is locked on $\lambda$ (solution 2), so would not accept a proposal that conflicts it. From this it follows that in no view from $v$ onwards will a log conflicting with $\lambda$ be committed.

### 2.2.3 Optimistic responsiveness

A system is responsive if it is able to make progress as fast as conditions allow when a non-faulty leader is elected. If a system must wait for some timeout $\Delta$ to elapse before making progress

1. New-view:
   (a) leader → replicas: “view = 3"
   (b) replicas → leader: “view = 2, log = [‘hello’, ‘world’]", ...
   (c) leader: waits $\Delta$ to hear from all non-faulty replicas
2. Prepare:
   (a) leader → replicas: pre-propose “log = [‘hello’, ‘world’, ‘!’]"
   (b) replicas verify the pre-proposal does not conflict with their *lock*
   (c) replicas → leader: “log = [‘hello’, ‘world’, ‘!’] ack", ...
3. Commit (lock):
   (a) leader → replicas: propose “log = [‘hello’, ‘world’, ‘!’]", qc = (*prepare* acks from previous stage)"
   (b) replicas *lock* on the proposed log
   (c) replicas → leader: “ack", ...
4. Decide:
   (a) leader → replicas: “decide, qc = (*commit* acks from previous stage)"
   (b) replicas: execute log

Figure 2.2: Example of byzantine fault-tolerant consensus algorithm.

then it is not responsive.

The algorithm demonstrated in figure 2.2 is not responsive. In the *new-view* phase the leader must wait to hear from $n - f$ replicas, and for a timeout of $\Delta$ to elapse. The timeout is necessary for the system to have the liveness property, so that it can always make progress when a non-faulty leader is elected. Consider again the *prepare* phase; the leader includes in its pre-proposal the QC from the highest view that it hears about in the *new-view* messages it receives. However, there may be some honest replica [how???] that the leader did not hear from (perhaps their message was lost) that is locked on a higher-view proposal than the one that the leader chooses to pre-propose. When this replica receives the pre-proposal, it will reject it as it is locked on a higher-view proposal, and the system will not make progress. This is why it is necessary to wait for a timeout $\Delta$, so that the leader waits a sufficient amount of time to receive a *new-view* from all replicas.

In order to achieve responsiveness we can modify our algorithm by adding a *pre-commit* phase in between *prepare* and *commit*, and removing the requirement for the leader to wait for a timeout in the *new-view* phase. As before, there may be some honest replica that becomes locked on some proposal $x$ during the *commit* phase. However, now there is a *pre-commit* phase before this, where a quorum of replicas all store a *key* for $x$, with this *key* being a QC of *prepare* acks. We now consider the next *new-view* phase; the replicas now send their *key* to the new leader, and the leader chooses what to propose based on the highest view *key* it sees. Even if the leader does not receive a *new-view* from the honest replica itself, it is guaranteed to hear about the proposal $x$, so it will be able to make progress. See an example in figure 2.3.

1. New view:
   (a) leader → replicas: "view = 3"
   (b) replicas → leader: "qc = (*key* for view = 2, log = ['hello', 'world'])", ...
2. Prepare:
   (a) leader → replicas: pre-propose "log = ['hello', 'world', '!'], qc = (key for view = 2, log = ['hello', 'world'])"
   (b) replicas verify the pre-proposal does not conflict with their *lock*
   (c) replicas → leader: "log = ['hello', 'world', '!'] ack", ...
3. Pre-commit (key):
   (a) leader → replicas: "qc = (*prepare* acks from previous stage)"
   (b) replicas store qc as a *key*
   (c) replicas → leader: "log = ['hello', 'world', '!'] ack", ...
4. Commit (lock):
   (a) leader → replicas: propose "qc = (*pre-commit* acks from previous stage)"
   (b) replicas *lock* on proposed log
   (c) replicas → leader: "ack", ...
5. Decide:
   (a) leader → replicas: "decide, qc = (*commit* acks from previous stage)"
   (b) replicas: execute log

Figure 2.3: Example of responsive consensus algorithm.

### 2.2.4 View-changes

If a leader fails to make progress within some timeout a view-change takes place and the next view begins. The HotStuff paper does not go into detail on how view-changes take place, so this explanation is based on a talk by one of its authors, Ittai Abraham [1].

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.

The use of timeouts in this way is a type of failure detector, which is a system that facilitates the detection of failed nodes. [add more information on failure detectors ***]

### 2.2.5 Chaining

The algorithm presented goes through several very similar phases to commit a proposal, these phases involve collecting votes from replicas to form a QC that then serves in later phases. The 'chained' algorithm simplifies and optimises this by pipelining the different phases. Instead of having different phases as before, we can have a single *generic* phase that collects votes, creates a *generic QC*, and sends it to the next leader; now we change view on each phase and a QC can serve in multiple phases concurrently.

In each view, some leader sends a proposal to all replicas, who send their replies to the next leader who can form a QC to add to their proposal. The replicas also send *new-view* messages to the next leader as before, to allow them to select a node to propose.

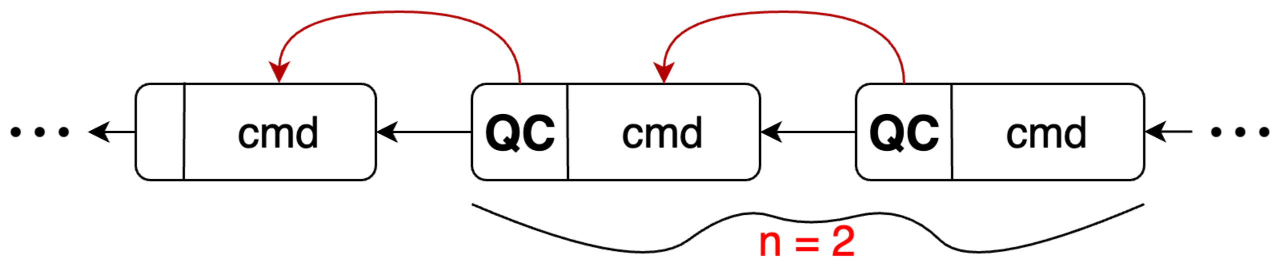


Figure 2.4: Sequence of nodes forming a 2-chain.

Figure 2.4 shows a chain of nodes connected by 'parent' links; every time we propose a value we extend the chain with a new node. Some node $b$ also contains a link to another node within *b.justify.node*, which is the previous generic QC in the chain. In the event of a view-change a dummy node will be inserted in the chain which will cause a 'gap' where some *b.justify.node* pointer jumps over a dummy node. If some node $b$ has a QC that points to its direct parent with no 'gap' in-between, then we say that it forms a *one-chain*. In general, if we have $n$ such direct links without 'gaps' we refer to this as an *n-chain*, as shown above.

To reproduce the behaviour of the unchained algorithm, on receiving a proposal for node $b^*$ a replica must look at $b^*$*.justify.node* to see if it points to an *n-chain*. It must take different actions depending on the length of the chain:

- zero-chain: this is equivalent to the *prepare* phase from the unchained version, so the replica votes for the proposal only if it does not conflict with their *lock*
- one-chain: this is equivalent to the *pre-commit* phase from the unchained version, so the replica should store a *key*

- two-chain: this is equivalent to the *commit* phase, so the replica should *lock* on the value proposed at the start of the two-chain
- three-chain: this is equivalent to the *decide* phase, so the replica can execute the commands from the node at the start of the three-chain

## 2.3 Tools & libraries

*In this section we outline the languages and libraries used in implementation, and justify why they were appropriate for this project.*

### 2.3.1 OCaml

I chose OCaml for this project due to its high-level nature, static type system, ability to blend functional and imperative paradigms, and good library support. 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. Additionally, the Tezos cryptocurrency is written in OCaml and contains a cryptography library that provides the functionality needed by HotStuff.

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 has a powerful module system that facilitates writing highly reusable code. The module system was only briefly touched upon in the tripos (in Concepts in Programming Languages from IB), so I spent time learning about these features. Modules provide an elegant interface for the core state machine to interact with the imperative parts of the program that actually send messages over the network.

There is no existing reference implementation of HotStuff in OCaml, so my project contributes to the growing OCaml ecosystem. This ecosystem is home to an active community, and interesting projects such as MirageOS unikernels. Because my project is implemented purely in OCaml, it could potentially be deployed on a MirageOS unikernel [???].

### 2.3.2 Lwt

Lwt 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 and are used by asynchronous promises in Lwt. Lwt is useful to this project as promises provide a way to 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 [citation needed***].

One alternative I could have used is Jane Street's Async library [citation needed???], which has similar features but better performance. I chose not to use this library due to its poor documentation. Another alternative that has better performance than Lwt is EIO [citation needed???], but this library is new and not yet in a stable state.

### 2.3.3 Cap'n Proto

Cap'n Proto is an RPC framework that includes a library for sending and receiving RPCs, 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.

[some people use xyz fancy library that is much faster but this was clearly not feasible for this project. . . ]

### 2.3.4 Tezos cryptography

The Tezos cryptography library 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. 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. Benchmarks for the library are presented in section 4.2.2.

## 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 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

*In this section we describe the professional software engineering methodology deployed during implementation.*

### 2.5.1 Development methodology

We chose to use a waterfall model for this project, tasks were carried out in accordance with the timetable set out in the proposal [reference appendix!]. Development proceeded in a sequential fashion, we set out requirements (section 2.4), studied the HotStuff protocol (section 2.2), implemented the system in accordance with the timetable (section 3), and carried out an evaluation of performance (section 4).

### 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[1] of the consensus state machine code, the coverage report is at *_coverage/index.html*.

The Memtrace library and viewer 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.

[talk about mininet!]

Due to the distributed nature of the program, normal debugging tools and profilers are not useful for debugging deadlocks and performance issues. This is because the cause of deadlocks and performance issues is often some process waiting or a backlog of work forming, but this cannot be detected by tools that just track things like CPU usage. Instead, I had to rely on manual inspection of the program trace and commands that measure the real time taken for some part of the program to run.

[CI???]

### 2.5.3 Source code management

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

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

Chapter 3

# Implementation

*In this chapter we describe the architecture of our implementation [3.1], present a specification and proof of correctness and liveness for our pacemaker [3.2], describe key optimisations implemented [3.3], present our load generator and experiment scripts which will be used in evaluation [3.4], and give an overview of the repository structure [3.5].*

## 3.1 Architectural overview

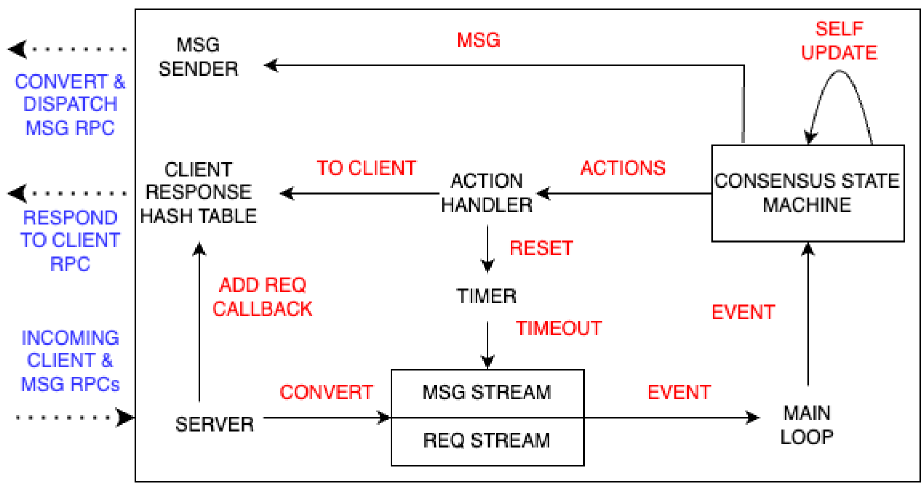


Figure 3.1: Architecture of a node.

The core implementation of the HotStuff algorithm (section 3.2) is implemented in the *consensus* module. The main function provided by this module is *advance*, which delivers some event (such as an incoming message, client request or timeout) to the consensus state machine and returns an updated state and a list of actions (such as sending a message to another node or responding to a client request) to be carried out. This architecture is inspired by the OCons project[1], which was developed by my project supervisor. The *consensus* module contains both a chained and unchained implementation that share a common 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.

[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.

Each node operates as a server waiting for messages from other nodes or requests from a client (or in the case of our experiments a load generator, which is described in section 3.4). The format of RPCs is specified in a Cap'n Proto schema, in their custom markdown language [*** cite instead of link!]. A received RPC must be decoded, and the Cap'n Proto types converted into the internal types of the consensus state machine. Messages and requests are added to separate streams[2] when they are received. When a request is received the callback function to respond to the request is stored in a hash table, so that it can be accessed when the command has been committed and the client request can be responded to.

The main loop takes events from the message and request streams, prioritising internal messages over client requests [expand on why, maybe cite something ***]. It takes these events and delivers them to the *advance* function of the consensus state machine. This architecture was chosen so that the *advance* function is never run in parallel on different messages / requests, as this could lead to race conditions. The *advance* function then returns a new state which is stored, and a list of actions to carry out.

The actions that can be carried out are sending a message to other nodes, responding to a client request, and resetting a timer. In order to send a message we must convert the consensus state machine internal types into Cap'n Proto types, and construct an RPC that matches the schema. Messages are dispatched asynchronously in a new thread. The node maintains TCP connections with all other nodes that are reused every time a message is sent, and in the event of the connection breaking the node repeatedly attempts to reconnect with binary exponential back-off. When responding to a client request the committed command's unique identifier is used to lookup the callback to respond to the client, which is then called, sending a response to the client. The timer is implemented with Lwt promises, a new thread is created which waits for a timeout to elapse and then adds a TIMEOUT message to the message stream.

## 3.2 Pacemaker Specification

We present the pseudocode for the unchained algorithm given in the HotStuff paper (algorithms 1 & 2) with our additions coloured in green and modifications in pink. We have only shown the main changes and not the other features and performance improvements we have made (such as batching), which are presented in section 3.3. In order to better map to the original pseudocode we break the algorithm into HotStuff and the pacemaker, although in our modified algorithm these two sections are not cleanly separable. We informally justify the changes, and then prove that they yield the desired behaviour.

The model of collecting a quorum of COMPLAIN messages to transition to the next view is based on the pacemaker for LibraBFT [cite], and is described in section 2.2.4. The rest of the pacemaker is designed to advance to the next view as soon as the system allows, and is based on ad-hoc fixes to deadlocks that were encountered during implementation.

- CREATELEAF (algorithm 1, line 1): This function has been modified so that 'dummy' nodes are inserted to maintain the invariant that the height of the chain is always one greater than the current view number, as in the non-event driven pseudocode in the original paper.

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

**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(MSG$_v$(GENERIC, $b_{new}$, $qc$))
22: **if** $v =$ GETLEADER($m.view$) $\wedge\ m.view = curView$ **then**
23: **if** $b_{new}.height > vheight \wedge (b_{new}$ extends $b_{lock} \vee n.height > b_{lock}.height)$ **then**
24: $vheight \leftarrow b_{new}.height$
25: SEND(GETLEADER(), VOTEMSG$_u$(GENERIC, $b_{new}$, $\perp$))
26: UPDATE($b_{new}$)
27: **if** not ISNEXTLEADER() **then**
28: ONNEXTSYNCVIEW($curview + 1$)
29: **procedure** ONRECEIVEVOTE(VOTEMSG$_v$(GENERICACK, $b$, $\perp$))
30: **if** ISLEADER($m.view + 1$) $\wedge\ m.view \geq curView$ **then**
31: **if** $\exists(v, \sigma') \in V_{\text{m.view}}[b]$ **then**
32: **return**
33: $V[b] \leftarrow V_{\text{m.view}}[b] \cup \{(v, m.partialSig)\}$
34: **if** $|V_{\text{m.view}}[b]| \geq n - f$ **then**
35: $qc \leftarrow QC(\{\sigma | (v', \sigma) \in V_{\text{m.view}}[b]\})$
36: UPDATEQCHIGH($qc$)
37: ONNEXTSYNCVIEW($m.view + 1$)
38: **function** ONPROPOSE($b_{leaf}, cmd, qc_{high}$)
39: $b_{\text{new}} \leftarrow$ CREATELEAF($b_{leaf}$, $cmd$, $qc_{high}$, $b_{leaf}.height + 1$)
40: BROADCAST(MSG$_v$(GENERIC, $b_{new}$, $qc_{high}$))
41: **return** $\text{b}_{\text{new}}$

- ONRECEIVEPROPOSAL (algorithm 1, line 21): The change on line 22 ensures that replicas only respond to a proposal from the leader of the current view. The change on line 27 means that a replica transitions to the next view once it receives a proposal unless it is the next leader, in which case it waits to collect the VOTEMSGs before transitioning.
- ONRECEIVEVOTE (algorithm 1, line 29): The change on line 30 ensures that a replica

**Algorithm 2** Modified Pacemaker

1: **function** GETLEADER<br>
2:     **return** $curView$ mod $nodeCount$<br>
3: **procedure** UPDATEQCHIGH($qc'_{high}$)<br>
4:     **if** $qc'_{high}.node.height > qc_{high}$ **then**<br>
5:         $qc'_{high} \leftarrow qc_{high}$<br>
6:         $b_{leaf} \leftarrow qc'_{high}.node$<br>
7: **procedure** ONBEAT($cmd$)<br>
8:     **if** $u =$ GETLEADER() **then**<br>
9:         $b_{leaf} \leftarrow$ ONPROPOSE($b_{leaf}$, $cmd$, $qc_{high}$)<br>
10: **procedure** ONNEXTSYNCVIEW($view$)<br>
11:     $curView \leftarrow view$<br>
12:     RESETTIMER($curView$)<br>
13:     ONBEAT($cmds.take()$)<br>
14:     SEND(GETLEADER(), $\text{MSG}_u$(NEWVIEW, $\perp$, $qc_{high}$))<br>
15: **procedure** ONRECEIVENEWVIEW($\text{MSG}_u$(NEWVIEW, $\perp$, $qc'_{high}$))<br>
16:     UPDATEQCHIGH($qc'_{high}$)<br>
17: **procedure** ONRECIEVECLIENTREQUEST(REQ($cmd$))<br>
18:     $cmds.add(cmd)$<br>
19: **procedure** ONTIMEOUT($view$)<br>
20:     SEND(GETNEXTLEADER(), MSG(COMPLAIN, $\perp$, $\perp$))<br>
21:     RESETTIMER($view + 1$)<br>
22: **procedure** ONRECIEVECOMPLAIN($m =$ MSG(COMPLAIN, $\perp$, $\perp$))<br>
23:     **if** ISLEADER($m.view + 1$) $\wedge\ m.view \geq curView$ **then**<br>
24:         **if** $\exists(v, \sigma') \in C_{\text{m.view}}[b]$ **then**<br>
25:             **return**<br>
26:         $C_{\text{m.view}}[b] \leftarrow C[b] \cup \{(v, m.partialSig)\}$<br>
27:         **if** $|C_{\text{m.view}}[b]| = n - f$ **then**<br>
28:             $qc \leftarrow QC(\{\sigma | (v', \sigma) \in C_{\text{m.view}}[b]\})$<br>
29:             BROADCAST(MSG(NEXTVIEW, $\perp$, $qc$))<br>
30: **procedure** ONRECEIVEANY($m =$ MSG($*$, $*$, $qc$))<br>
31:     **if** $qc.view \geq curView$ **then**<br>
32:         ONNEXTSYNCVIEW($qc.view + 1$)

will ignore messages from earlier views, and it checks that it is the correct destination for a vote message. Another change we make is dividing $V$ into different sets for messages from different views; this prevents votes from different views being used to form a quorum. The change on line 37 means that once a leader has received a quorum of messages, it can transition to the next view.

- ONPROPOSE (algorithm 1, line 38): The change on line 40 includes a QC along with the proposal, allowing replicas to catch up if they are in a lower view.
- GETLEADER (algorithm 2, line 1): The original pseudocode states that this function is application specific. We have chosen to use a round-robin system to assign leaders to views.
- ONNEXTSYNCVIEW (algorithm 2, line 10): This function previously just sent a NEWVIEW message to the next leader. Our modified function also updates the $curView$, and resets

the timer for the new view. Additionally it calls ONBEAT which causes a leader to propose a new value.

- ONRECIEVECLIENTREQUEST (algorithm 2, line 17): On receiving a client request we simply add it to a queue of commands waiting to be proposed.
- ONTIMEOUT (algorithm 2, line 19): When the current view times out replicas will send a COMPLAIN to the next leader, and reset their timers for the next view. This behaviour is explained in 2.2.4.
- ONRECIEVECOMPLAIN (algorithm 2, line 22): This function is very similar to ONRECEIVEVOT except that a quorum of COMPLAIN messages is being collected rather than votes. On achieving a quorum the leader can broadcast a NEXTVIEW message to get the replicas to transition to the next state, including the quorum of COMPLAINs as proof. This behaviour is explained in 2.2.4.
- ONRECEIVEANY (algorithm 2, line 30): N.B. the message type given is the wildcard operator, so this procedure is run on every message we receive. The function checks if the quorum received is from a greater view than *curView*, if so then it is safe to transition to that view in order to catch up.

### 3.2.1 Proofs

We now present a reasonably formal proof that the modified pacemaker possesses qualities that ensure the liveness of the system. The first property (theorem 3.2.1) ensures that there will always be opportunities for the system to make progress, and the second that the pacemaker cannot be controlled by a byzantine node (theorem 3.2.4). These properties are presented by xyz in the Cogsworth paper [cite!!!], which formalised the notion of a pacemaker and what qualities it should possess.

HotStuff works under a partially-synchronous system model (section 2.2), so the following proofs assume that global synchronisation time has been reached, and messages have a bounded latency of $\delta$.

**Theorem 3.2.1** (View synchronisation)**.** *There exists infinite views with honest leaders that all honest replicas will be in simultaneously, and have enough time to make progress.*

*Proof.* From lemma 3.2.2 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 3.2.3 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 37) — $l_1$ 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.

Once all honest replicas are in a view with an honest leader, they will only transition once they have made progress, or once they timeout, which shouldn't happen if the timeout is sufficiently long. Hence they will all be in the view long enough to make progress. □

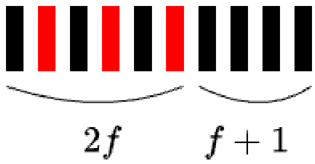


Figure 3.2: Example of round-robin leader allocation for $f = 1$. Red rectangles denote byzantine leaders.

**Lemma 3.2.2.** *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.*

*Proof.* We use a round-robin system to allocate leaders to views. If we attempt to alternate honest and byzantine leaders, there will always be $f + 1$ consecutive honest leaders left over (figure 3.2). Hence there will always be at least 2 consecutive honest leaders (the lemma holds trivially for $f = 0$). □

**Lemma 3.2.3.** *An honest replica $x$ will eventually enter a view where it is leader.*

*Proof.* In the event that a byzantine node is leader and 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. □

**Theorem 3.2.4** (Synchronisation validity)**.** *The pacemaker will only advance the view if at least one honest consensus machine*[3] *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. □

## 3.3 Performance improvements

*In this section we highlight the key performance optimisations we implemented, and describe some of the debugging and experimentation that led to them. [this is a hard problem, reference chubby again. bugs are subtle.]*

[3]For this proof we consider the consensus machine (algorithm 1) and the pacemaker (algorithm 2), to be separate entities.

### 3.3.1 Batching

One way to improve goodput (number of requests committed) is to 'batch' requests, meaning a node may contain many commands instead of just one. This can dramatically increase goodput as now a single view can result in many commands being committed and executed instead of just one.

In order to implement this change in algorithm 2 one simply has to modify the ONNEXTSYNCVIEW procedure. Instead of taking a single element from the queue of commands waiting to be proposed, the whole queue will be 'batched' into a single proposal.

In theory this change should result in much higher goodput without any significant increase in latency. Analysis of timing data after implementing this feature revealed that latency had increased substantially. We now present some of the analysis and experimentation that was carried out to diagnose this issue. As mentioned in 2.5.2, the nature of the project meant that debugging had to be carried out by manual inspection of the program trace and timing sections of the program. To overcome this I carried out tests in a scientific manner, constructing a hypothesis for why the program was slow based on crawling through logs, then attempting to test my hypothesis while controlling other variables, and finally implementing a solution.

#### Probing effects

I added timing and logging statements to key parts of the program, allowing me to better diagnose the source of the poor performance. Running a live test with print statements has the advantage that one can quickly see when progress is not being made, or when there are pauses, as the print statements stop. However, this benefit is outweighed by the sheer volume of logs and times being printed, it becomes difficult to manage when logs are in the millions of lines long.

Moreover, debugging by print statements had a bigger problem of inconsistencies between runs making it difficult to diagnose any problem. This was because the large volume of print statements being executed had a significant effect on the performance of the program. This is known as a *probing effect*, the behaviour of a system is altered by the act of measuring it. I initially assumed that because the print statements were not in-between the timing statements they would not affect my measurements, but they are an expensive operation that can cause delays to happen in execution where one would not expect [cite???].

This problem was overcome through the development of a simple logging framework. Key parts of the program such as the state machine advancing, actions being carried out, messages being sent, are all timed and added to lists. Critically, this list is stored and not printed out until the node is killed, so the printing cannot interfere with execution. Relevant statistics such as mean, standard deviation, and total time for each interval measured are outputted, providing crucial insight into the performance of the program.

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

**Architectural experimentation**

Initial analysis of timing data showed a large amount of time was spent by requests queuing on the node before they are delivered to the state machine. At this time the architecture presented in section 3.1 had not been fully developed, specifically, there was only one stream that contained both internal messages and client requests.

**Hypothesis:** Internal messages are being starved by client requests. At large throughputs the stream quickly becomes filled with client requests, which may prevent internal messages being handled. Internal messages represent a backlog of work that is not yet finished, so this should be handled before accepting more requests.

**Design principle:** The system should prioritise clearing its backlog of work before accepting new work.

**Potential solution:** Split the stream into two separate streams for messages and requests, and always pick from the message stream over the request stream until the message stream is empty.

Implementing the potential solution did not lead to a significant improvement in performance, so this was not the issue. However, this architectural model was retained as it is theoretically better and may lead to observable improvements later.

Another potential cause of the issue was that internal messages are starving client requests, rather than the other way around. The modified implementation of the pacemaker (3.2) immediately begins a new view as soon as the previous one is finished, which causes a large amount of internal messages.

**Hypothesis:** Because the state machine is constantly advancing at the fastest possible rate, the volume of internal messages may prevent new requests from being handled.

**Experimentation:** One approach is to 'balance' the starvation by mostly prioritising internal messages, but every $x$ iterations picking a request instead of an internal message. Varying $x$ led to significant changes in performance, but no value for $x$ led to a good level of performance. This approach did not appear workable, as the nature of starvation seemed to be a complicated, with messages starving requests, and the other way around.

**Potential solution:** Instead of starting the next view whenever possible, deliver the *BEAT* and *ONNEXTSYNCVIEW* interrupts at a steady rate.

This potential solution was implemented; it involved checking a timer on every iteration of the main loop to see if some $\Delta$ had elapsed and it was time to deliver the next interrupt, as well as significant modifications to be made to the consensus state machine. Implementing this feature actually led to a significant performance hit, so the changes were discarded. However, experimenting with this feature showed an interesting phenomenon; the main loop, which is infinitely recursive, was running at a much slower rate than one would expect. Further timing statements revealed that the command which removed an element from the stream would sometimes take a very long time (on the order of seconds). Reading more into the Lwt documentation led to the following idea:

**Potential solution:** Use a different method to take elements from the queue that is synchronous. This should stop the main loop blocking waiting for the queue to fill up.

This change improved performance by reducing the amount of time spent waiting in the main loop.

### Send-to-all and filtering

Further problems were uncovered by the implementation of send-to-all in the load generator (section 3.4). Without this feature a client would randomly chose a node to send their request to, and the command would not be proposed until the round-robin system reached that node to become the leader. Send-to-all can reduce latency by instead broadcasting a request to all nodes, meaning that the next leader can propose the command without having to wait for the round-robin system to reach them.

Instead of reducing latency as expected the implementation of this feature actually led to dramatically increased latency, and some requests not being fulfilled at all within the time that the experiment ran. Heatmaps that show the distribution of latencies over the course of an experiment showed that latency increased exponentially over the course of the test.

**Hypothesis:** Send-to-all overloads the system by increasing the number of requests by a factor of $n$, where $n$ is the number of nodes.

**Experimentation:** If our hypothesis is true, then we should see similar behaviour in a send-to-one system run at $n$ times the throughput. This is indeed the case; running a send-to-one system with such high throughput led to exponential growth in latencies.

**Further evidence:** Implementing this feature resulted in Lwt errors being thrown. As described in 3.1, in order to respond to client requests, the node maintains a hash table of promises that can be awakened when the consensus state machine has successfully committed and can respond to the client. The Lwt error was caused by promises stored in the hash table being awoken multiple times, which causes an error. This provides further evidence that the same commands are being committed and responded to multiple times, which is wasted work.

**Potential solution:** Filter commands in the system to minimise the number of commands that are proposed by multiple nodes.

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

One way to implement this is to maintain a set of commands that have been committed, and before proposing or committing a node, calculating the set difference with the *commited* set to filter out redundant commands. Instead of nodes containing a list of commands, as in our naive implementation of batching, we now use a set to enable efficient computation of the difference.

This implementation did not lead to the desired reduction in latency, the system still exhibited the exponential growth previously observed. The lack of change in observable behaviour

**Algorithm 3** Filtering implementation #1

```
procedure OnCommit(b)
    if b_exec.height < b.height then
        toCommit ← b.cmds \ commited
        commited ← commited ∪ toCommit
        OnCommit(b.parent)
        Execute(toCommit)
procedure OnNextSyncView(view)
    curView ← view
    OnBeat(cmds \ commited)
    cmds = {}
    // ...
procedure OnRecieveClientRequest(Req(cmd))
    cmds ← cmds ∪ {cmd}
```

implied that the filtering of requests was not happening successfully.

**Hypothesis:** A command ($x$) is committed four views after it is proposed. In these four views our deduplication is ineffective in screening $x$ from being proposed again. Notably another node will not try to commit $x$ for a second time, but the data indicates that being proposed multiple times is a more important factor affecting performance.

**Experimentation:** Count the elements that are filtered out from being proposed by measuring the value $|cmds| - |toPropose|$. This value was small, which supports the hypothesis that filtering was largely ineffective.

**Potential solution:** Screen incoming commands more aggressively by instead maintaining a set of any commands that have been *seen*. Only filter out commands when they are being proposed rather than when they are being committed, as there is no evidence that screening at commit time gives any tangible benefit.

**Algorithm 4** Filtering implementation #2

```
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}
```

Note the small optimisation on line 8, the *seen* can safely be emptied as the commands have already been filtered; this reduces the amount of computation required to calculate the set difference next time. The second implementation was much more effective at screening requests and resulted in a significant reduction in latency. The difference is demonstrated in an ablation study (section 4.3.3) The effectiveness of this change led to further insight into what the bottlenecks for performance were.

#### Batch sizes

The effectiveness of filtering out commands to make proposals smaller implies that the size of a proposal has a significant impact on the performance of the program. Furthermore, analysis of the time data for sending messages showed that there was a high variance in time taken, with some messages taking on the order of milliseconds [verify ***] to be sent.

**Hypothesis:** Large batches of commands cause messages to become larger, which causes them to be sent slowly due to limitations in the RPC framework.

**Potential mitigation:** Limit the size of batches to prevent messages becoming too large.

Implementing this feature required minimal changes to algorithm 4, one simply has to take a subset of *cmds* to propose instead of the whole set. This feature gave a significant performance improvement, and largely eliminated the exponential growth in latency that had been seen previously. There is a trade-off between sending larger batches (more commands are committed, but messages take longer to send) and sending smaller batches (less commands are committed, but messages are sent faster). This trade-off is explored more in our analysis of the system with different batch sizes [refer to evaluation***]. Fundamentally, there are limitations in the RPC framework that give an upper bound on the performance we can hope to achieve, these limitations are evident in our benchmarking of the Cap'n Proto framework (4.2.1).

### 3.3.2 Encoding of nodes

*This section concerns the difficulties of encoding nodes in the Cap'n Proto schema, and designing a suitable type to represent them. As mentioned in 3.1, Cap'n Proto requires a schema to define the format of RPCs that can be sent, and one must convert between consensus state machine internal types and Cap'n Proto types in order to send messages.*

#### Recursive types

The unchained HotStuff algorithm has a genesis node $b_0$ that starts the chain of nodes. Each node contains the fields *parent*, *cmd*, *justify*, and *height*, with the *justify* field containing a quorum certificate with a 'pointer' to another node in the chain (section 2.2.5). The genesis node $b_0$ contains a hardcoded link to itself, so $b_0.justify.node = b_0$.

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.

### Node offsets

During live testing, memory usage increased rapidly, the nodes quickly exceeded the amount of memory the system had, and their processes were killed by the OS. Profiling with Memtrace (section 2.5.2) showed that the functions for converting between Cap'n Proto types and internal consensus types was responsible for the rapidly increasing memory usage.

This problem arose from the internal types that were designed to represent nodes. As well as the *node* type, there was a *qc* type that was used as the type of the *node.justify* field since it is a quorum certificate. By defining types in this way the 'pointers' in *node.justify.node* were not actually pointers, they contained a significant part of the chain. In this way the chain contained many redundant copies of parts of it, resulting in a very bloated *node* object that was very expensive to convert.

A solution to this problem is to store an offset to a node inside the *justify* field rather than the node itself - making it more like a real pointer. 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, that 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.

### Node comparison

Memtrace profiling also showed that the equality function for nodes was memory intensive. The solution to this was including a *digest* field in the node that is a hash over all of the other fields. Notably we can compute this hash over the *digest* field of the parent node rather than recursing through the whole chain. This means that we can also compare two nodes by comparing their digests without having to recurse through each chain; the digests being equal cryptographically guarantees that the whole chains are equal.

### Node truncation

Our current approach sends the entire node inside internal messages, which becomes expensive as the chain grows. As highlighted in section 3.3.1, the size of messages being sent is a bottleneck in the system, making this particularly expensive.

In order to overcome this problem, one can truncate the node before sending it, cutting off the node's parent link at some chosen depth. The entire node can then be reconstructed at the receiver, the received node can be 'spliced' back together with $b_{exec}$, which contains the node up to the point that has been executed. However, we must ensure not to truncate the log too much, so that there is a gap between what we send and $b_{exec}$ at the receiver, leading 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 we use a TCP-style approach. We include a field containing the height of the $b_{exec}$ node to the *PROPOSE*, *NEWVIEW*, 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. We 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.

## 3.4 Implementing for evaluation

*In this section we describe the infrastructure that will be used to evaluate our system in chapter 4, including the scripting developed to automate running experiments.*

### 3.4.1 Load generator

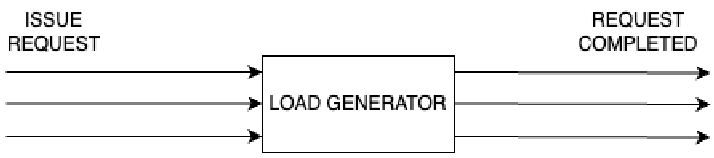


Figure 3.3: Open-loop load generator.

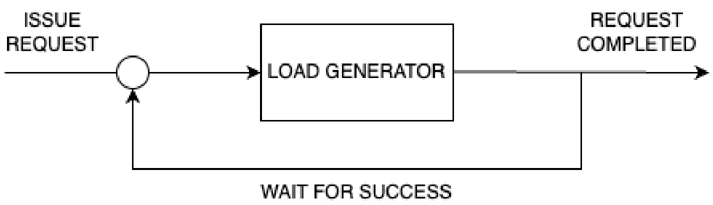


Figure 3.4: Closed-loop load generator.

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 a *kill* messages to the nodes, ending the test. It is also 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 by 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 both mean and standard deviation of latencies.

The load generator is open-loop (figure 3.3), 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 3.4), which must wait until it receives a response in order to send the next request. An open-loop load generator is more useful for our experiments as it allows us to overload the system and test its limits, whereas a closed-loop load generator 'waits' for the system, so cannot overload it.

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 (3.3.1) 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 we start the experiment, eliminating start-up effects.

### 3.4.2 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 all possible permutations of the input variables, duplicates them (so that each experiment is run multiple times to get better results), and runs them in a random order. By running experiments in a random order external factors that affect performance are somewhat mitigated. For example if the experiments were not run in a random order, the system may happen to experience interference when running experiments on 8 nodes, giving distorted results for this set of experiments. If instead the experiments were run randomly, the interference would affect some random group of experiments, and it would be more apparent that these results were anomalous. [could be more concise]

## 3.5 Repository Overview

- src
  - consensus
    - consensus_chained_impl.ml
    - consensus_impl.ml
    - consensus.ml
    - consensus.mli
    - crypto.ml
    - types.ml
    - types.mli
    - util.ml
  - experiments
    - data
    - graphs
    - plot.py
    - plotDummy.py
  - hs_api.capnp
  - api_wrapper.ml
  - api_wrapper.mli
  - dummy.ml
  - hs.ml
  - live_test.ml
  - main.ml
  - net.ml
  - types.ml
  - util.ml
  - runDummyExperiments.py
  - runExperiments.py
  - README.md

The *src* directory contains the main server loop and the code for interacting with Cap'n Proto to send messages. It also contains code for the load generator, and Python scripts to run experiments and benchmarks. Inside the *consensus* folder is the implementation of the consensus library based on the pseudocode presented in 3.2. N.B. this folder also contains testing files that are omitted for conciseness. The *experiments* folder is where the data from running experiments is outputted to, and it contains scripts for 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

*In this section we highlight the methods and hardware used in evaluation [4.1], benchmark the performance of Cap'n Proto and the Tezos cryptography library [4.2], and finally evaluate the performance of our HotStuff implementation [4.3]*

## 4.1 Testing methodology

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

Experiments were driven by an open-loop load generator (section 3.4.1), and were automated using Python scripts (section 3.4.2). In order to reduce the effect of interference, experiments were repeated 3 times, and the order of experiments was randomly permuted. In all experiments the load generator was run for 10 seconds, with a further 15 seconds after this without the load generator running to wait for any slow responses.

## 4.2 Library benchmarks

### 4.2.1 Cap'n Proto

[describe the experiment methodology, present argument, then back up with evidence from figures] We benchmarked the latency and goodput of sending messages in Cap'n Proto. We varied the size of messages sent in different tests to replicate the behaviour of the algorithm when sending 'batches' of many commands, so a message size of 600 means that the message size is approximately that of a message containing 600 commands.

The figures demonstrate that the framework has a severe drop in performance when sending large messages. For a message size of 600 the goodput goes to zero as the throughput increases, meaning that no messages are being responded to.

["Fundamentally, there are limitations in the RPC framework that give an upper bound on

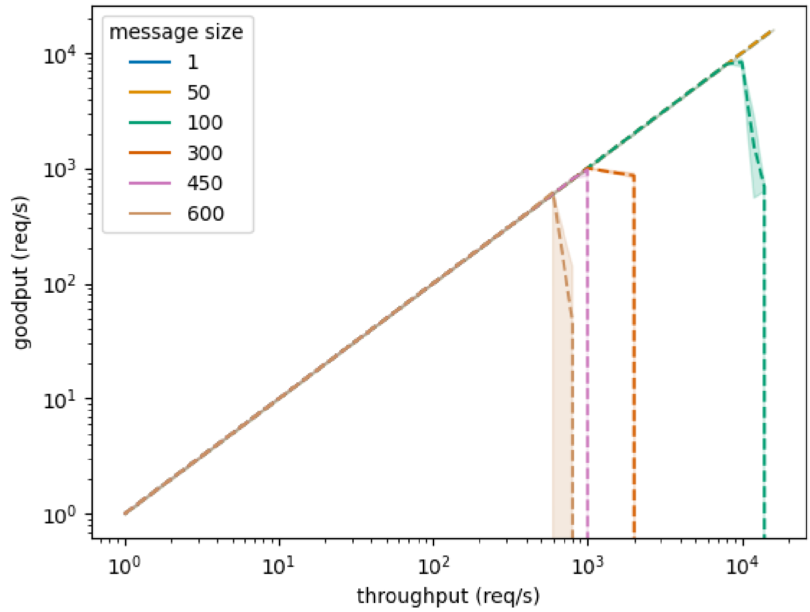


Figure 4.1: Benchmarking of Cap'n Proto server goodput for varying throughputs and message sizes.

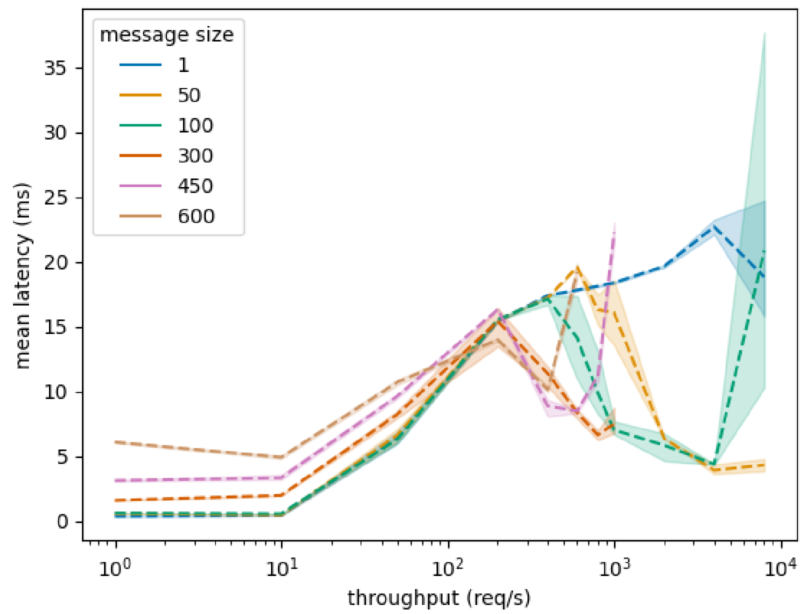


Figure 4.2: Benchmarking of Cap'n Proto server latencies for varying throughputs and message sizes. Discarded result if goodput was not within 5% of target throughput.

the performance we can hope to achieve, these limitations are evident in our benchmarking of the Cap'n Proto framework…"]

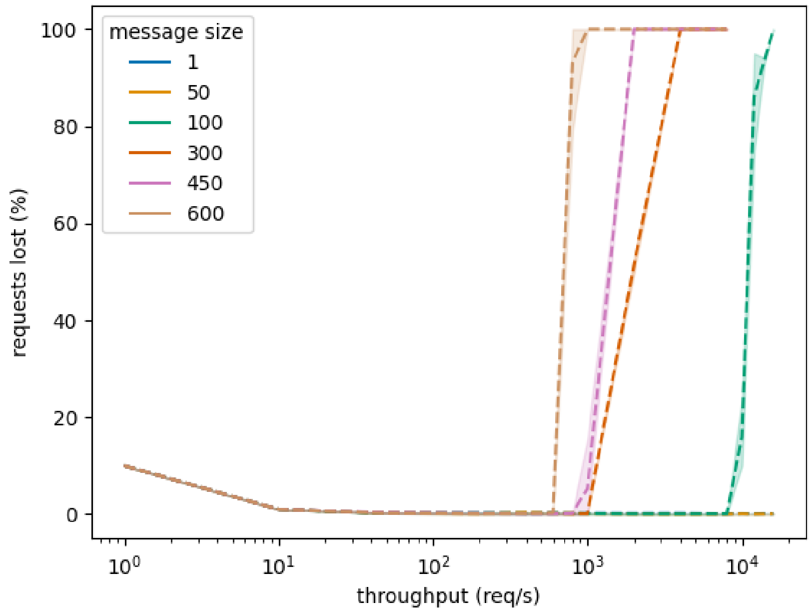


Figure 4.3: Benchmarking of Cap'n Proto server % of failed requests for varying throughputs and message sizes, run for 10s

### 4.2.2 Tezos Cryptography

I profiled the important functions of the library with Jane Street's Core_bench module [citation needed***]. 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.

| 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

We now analyse the performance and behaviour of the system with different parameters, and under different conditions. We argue that the optimisations described in section 3.3 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 cryptography (section 4.2.2).

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 4.4). When the

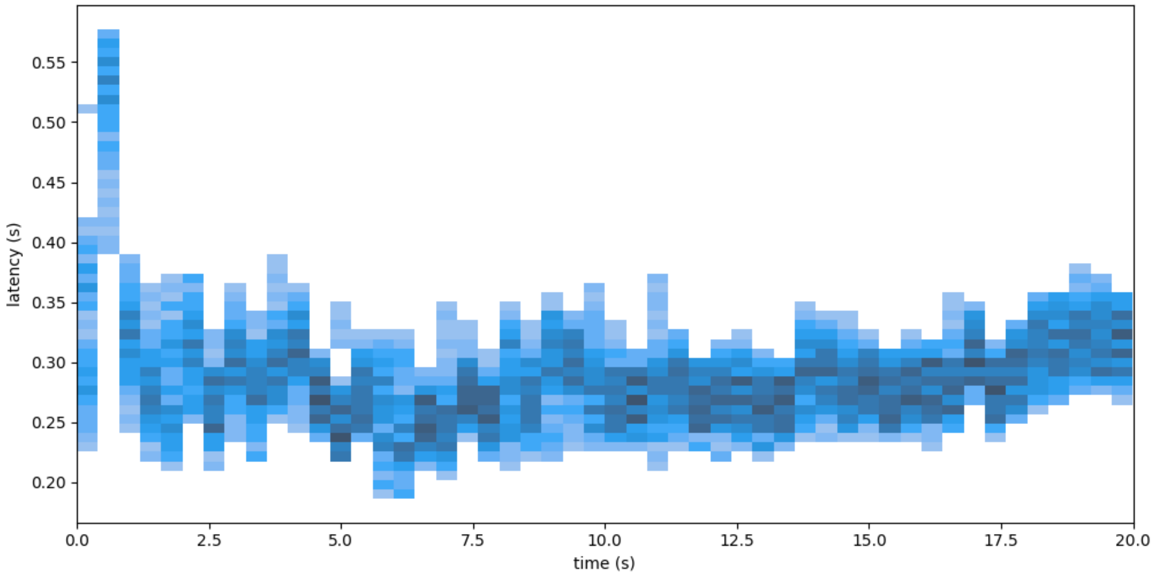


Figure 4.4: Heatmap showing relatively stable latency throughout the course of the experiment. [add parameters!!]

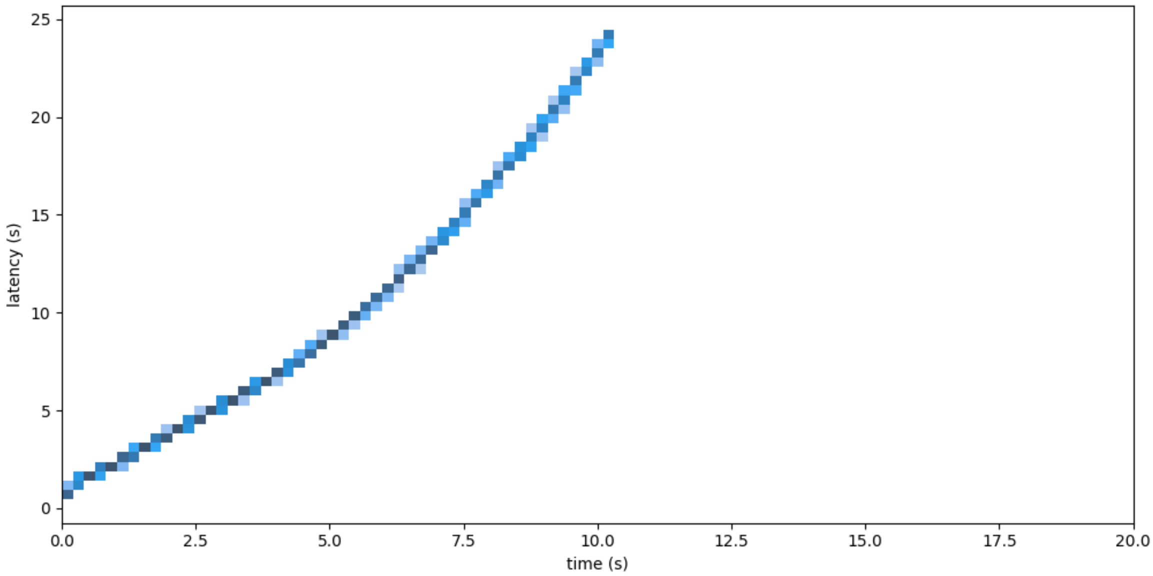


Figure 4.5: Heatmap showing linear growth in latency as the experiment progresses. [add parameters]

throughput exceeds the amount the system can keep up with, there is linear growth in latency as commands queue on the nodes (figure 4.5). 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.

In our comparison of batch sizes (section 4.3.1) there is evidence that the implementation of batching (section 3.3.1) was effective, as the system is able to 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, due to messages being larger and taking longer to serialise.

Our 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 means more messages must be signed and aggregated.

In our ablation study (section 4.3.3) we compare the performance of the system with different optimisations enabled, demonstrating their effectiveness in increasing goodput, and lowering latency. We also demonstrate that cryptography is a bottleneck by demonstrating the superior performance of the system with cryptography disabled.

In our wide area network (WAN) simulation study (section 4.3.4), we compare the performance of our system running locally, to a simulated mininet network (section 2.5.2) with link latency similar to what one might find in a wide area network. [we found...]

In our view-change study (section 2.2.4) we demonstrate that the view-change protocol (section 2.2.4) is effective in ensuring the system progresses once a node has died, albeit with a significant performance penalty.

### 4.3.1 Batch sizes

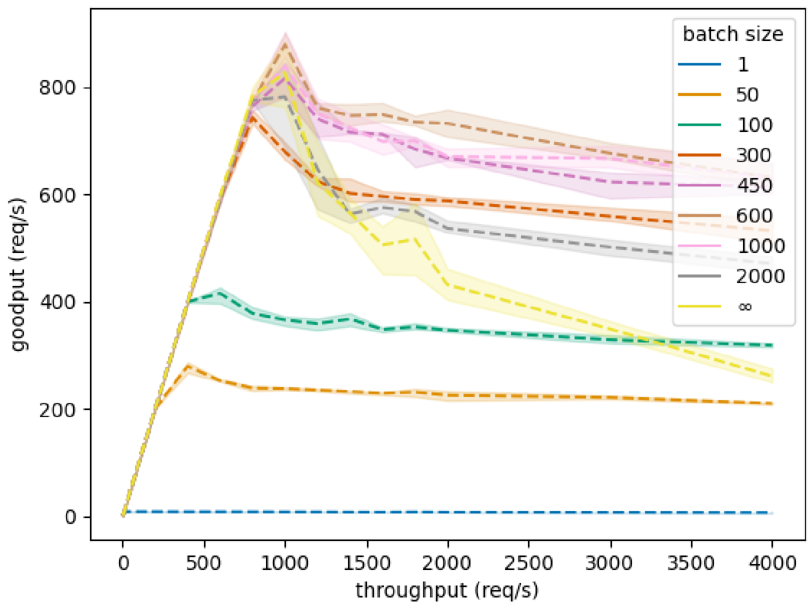


Figure 4.6: Benchmarking of goodput for varying throughputs and batch sizes.

This study compares the performance of the system for varying limits on batch sizes (section 3.3.1). All experiments were run on a network of 4 nodes.

At lower throughputs where the system is not overloaded, throughput grows linearly with goodput, as the system is able to responds to all incoming requests (figure 4.6), with roughly constant latency throughout an experiment (figure 4.4). During this period batches are not filled, so larger throughputs result in larger messages and a linear increase in latency due to increasing serialisation latency (figure 4.7). The system is able to reach a higher goodput before

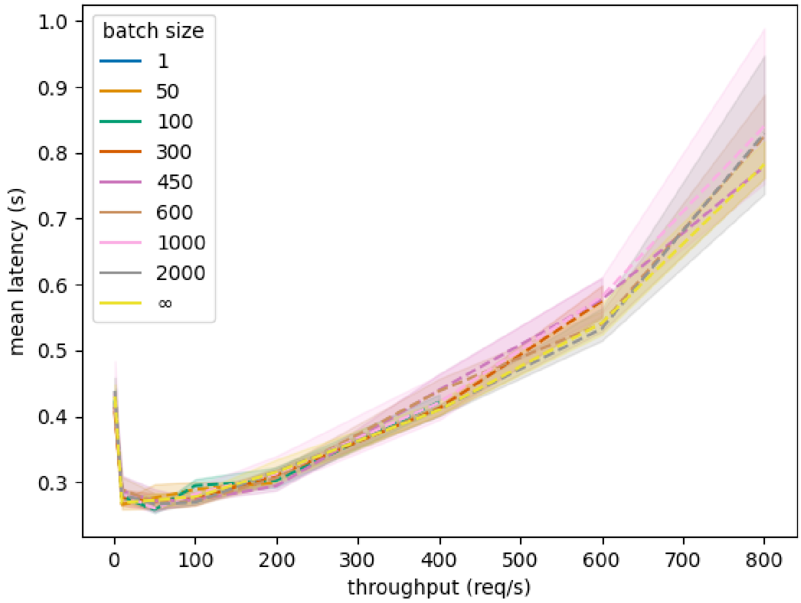


Figure 4.7: Benchmarking of mean latency while varying throughputs and batch sizes. Discarded result if goodput was not within 5% of target throughput.

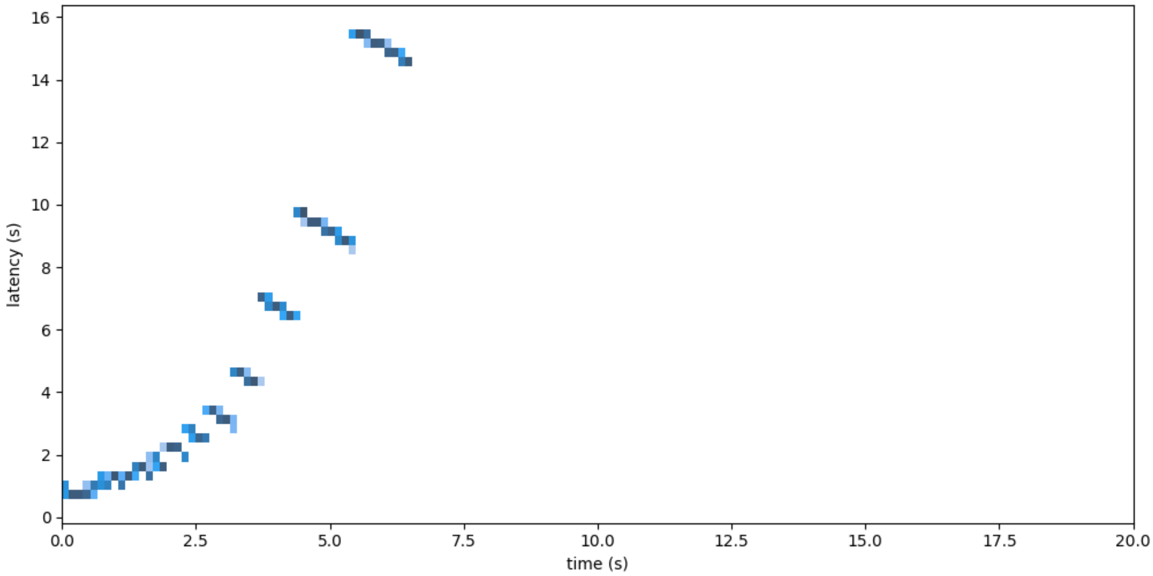


Figure 4.8: Heatmap showing exponential growth in latency as the experiment progresses. [add parameters]

being overloaded if it has a larger batch size, as each view results in more commands being committed; this supports our 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 4.6), as the system cannot handle the volume of requests; commands begin to queue on the nodes and latency increases linearly (figure 4.5). For higher batch size limits (especially unlimited), goodput declines once the system is overloaded rather than flattening off. This is because the benefits of larger batches are offset by messages becoming larger, causing increased serialisation latency, which increase 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 4.8.

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 our system appears to be a batch size of around 600 commands, with a maximum goodput of around 900req/s. (figure 4.6).

### 4.3.2 Node counts

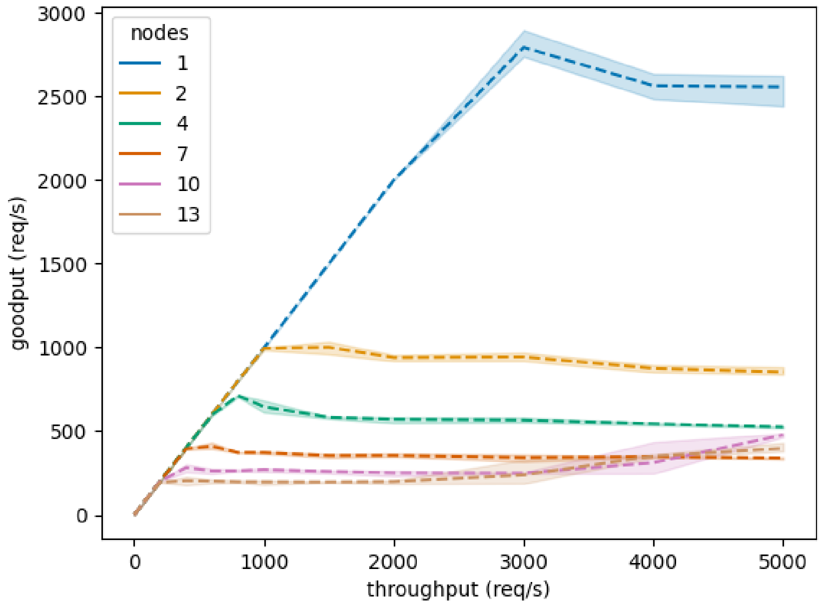


Figure 4.9: Benchmarking of goodput for varying throughputs and node counts.

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 with a batch size of 300.

Figure 4.10 shows that as node count increases, latency increases. This is because larger node counts mean that each view requires more internal messages to be sent to progress. Sending internal 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). As in our study of batch sizes (section 4.3.1), latency also increases linearly with throughput while the system is not overloaded. Notably the latency for a system of 1 node increases slowly, as there are no internal messages, just client requests and responses.

Figure 4.9 shows that the larger the node count, the lower the maximum goodput. This is again due to larger node counts resulting in more internal messages, causing more latency since

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

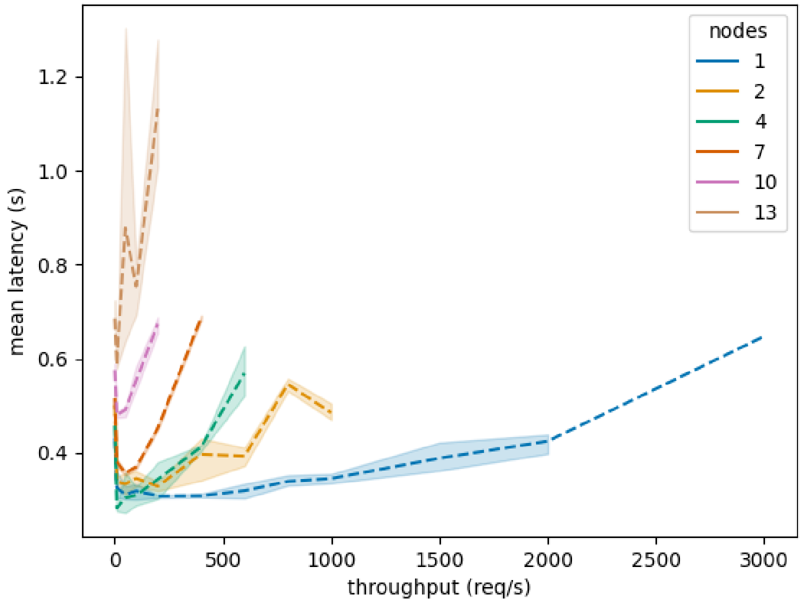


Figure 4.10: Benchmarking of mean latency while varying throughputs and node counts. Discarded result if goodput was not within 5% of target throughput.

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 | Chained | Truncation | Filtering | Crypto |
|---|---|---|---|---|
| 1 | ✗ | ✗ | ✗ | ✓ |
| 2 | ✓ | ✗ | ✗ | ✓ |
| 3 | ✓ | ✗ | ✓ | ✓ |
| 4 | ✓ | ✓ | ✗ | ✓ |
| 5 | ✓ | ✓ | ✓ | ✓ |
| 6 | ✓ | ✓ | ✓ | ✗ |

Table 4.2: Features enabled in different versions.

This study compares the performance of different versions of the system with different optimisations enabled. The optimisations explored are chaining (section 2.2.5), node truncation (section 3.3.2), and command filtering (section 3.3.1). We also compare performance with cryptography disabled. The mapping from version numbers to which features are enabled is given in table 4.2. All experiments were run with a network of 4 nodes, and unlimited batch sizes.

### 4.3.4 Wide area network simulation

[Give ablation graphs comparing the performance with 100ms delay, and without.]

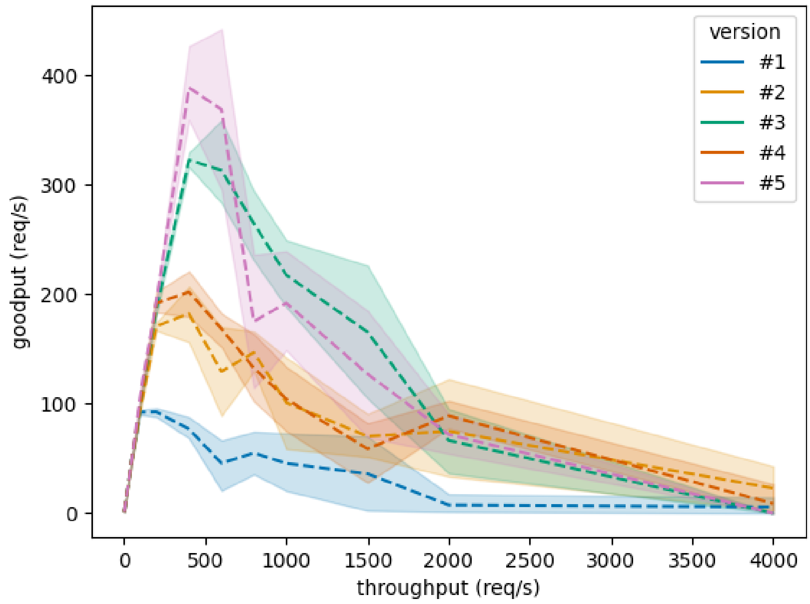


Figure 4.11: Benchmarking of goodput for varying throughputs and implementation versions, run for 10s with 4 nodes unlimited batch size.

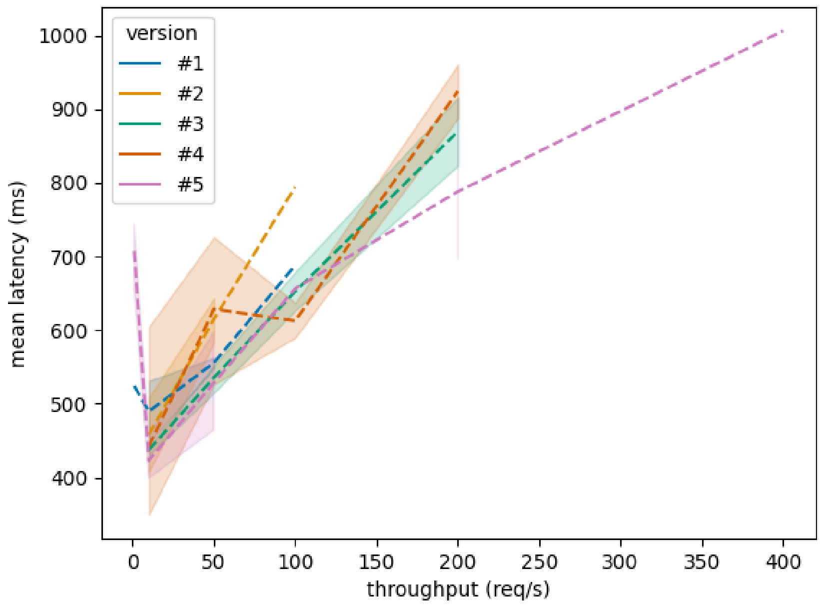


Figure 4.12: Benchmarking of mean latency while varying throughputs and implementation versions, run for 10s with 4 nodes unlimited batch size. Discarded result if goodput was not within 5% of target throughput.

### 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 2.2.4) is needed to skip the faulty leader's view and make progress.

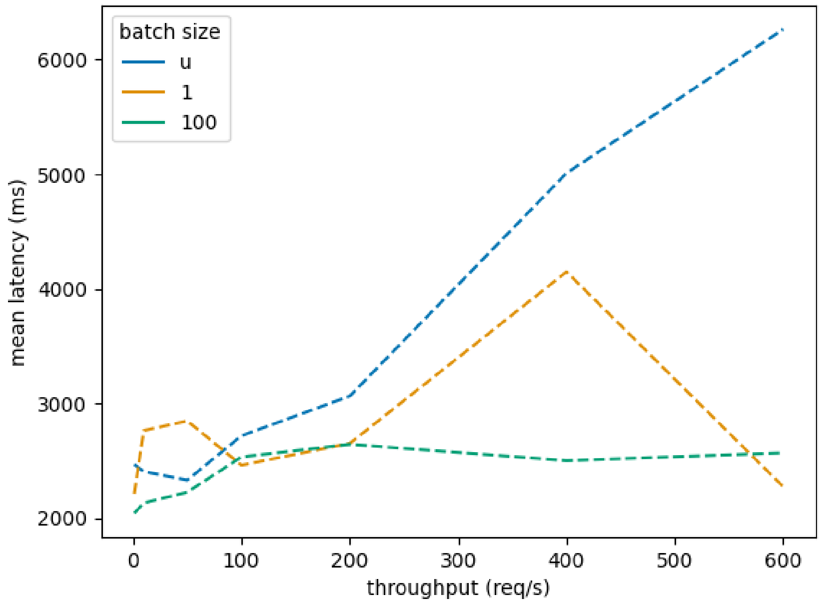


Figure 4.13: Benchmarking of mean latency while varying throughputs and batch sizes, run for 10s with 100ms network delay.

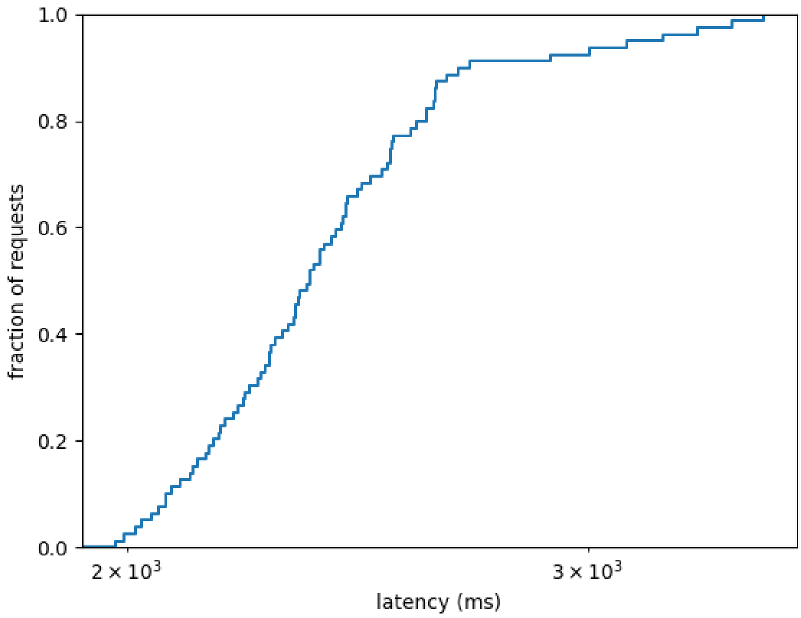


Figure 4.14: Cumulative latency plot for experiment with a throughput of 10req/s and unlimited batch size, run for 10s with 100ms network delay.

Figure 4.15 is a heatmap showing a node being killed 5s into an experiment. 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 around 1.25s is about what one would expect; 0.5s timeout and 0.75s of latency (the same as before the node was killed).

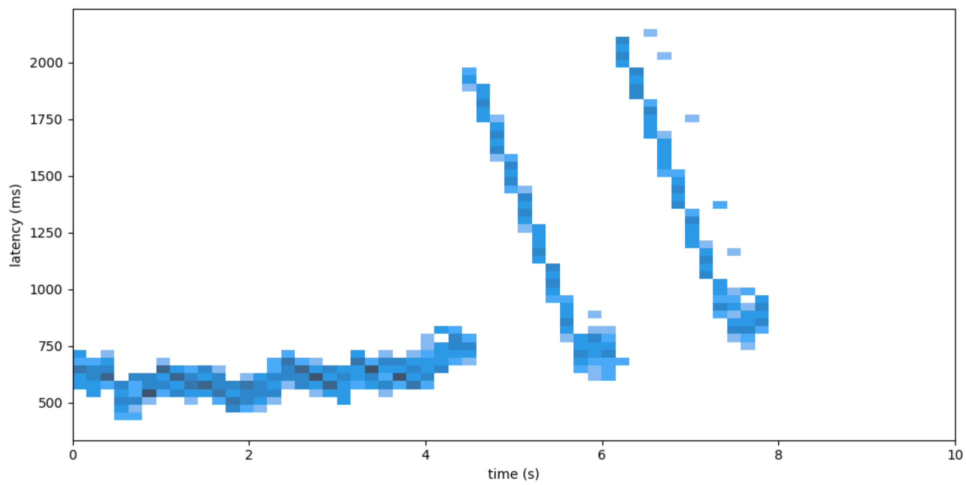


Figure 4.15: Heatmap showing distribution of latencies with a node being killed 5s in. Run for 10s with 7 nodes and a batch size of 100.

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 HotStuff protocol, although a failure-detector could help to minimise the number of views where the faulty node is leader.

## Chapter 5

# Conclusion

In this project we have given a reference implementation of the HotStuff byzantine consensus algorithm in OCaml, contributing to the wider OCaml ecosystem. The core algorithm is written in a self-contained module that could be reused by other projects with differing architectures and RPC systems. We have described some of the practical challenges of implementing HotStuff and implemented several optimisations of the basic algorithm (section 3.3). One main challenge was adapting the HotStuff pacemaker; we gave a full specification and proved that our pacemaker has desirable properties (section 2).

We have successfully met the requirements set out in section 2.4. There is significant evidence for the correctness of our implementation; our 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 we 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. We also implemented several optimisations (section 3.3), 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 [cite]; 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 our 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 highly non-trivial; some of our 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. Once could also implement a proof of work or proof of stake mechanism on top of our implementation to make it resilient to Sybil attacks, allowing it to be deployed in a permissionless setting.

One lesson I learnt for this project is that a more iterative development methodology, such as rapid application development, would have been more suitable for implementing a distributed algorithm than the waterfall model (section 2.5.1). Much of development time was spent debugging system performance to implement the optimisations described in section 3.3. 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 I would advocate an iterative model that interleaves development with evaluation, and 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] Abraham, I. Byzantine fault tolerance, state machine replication and blockchains.

[2] Chandra, T., Griesemer, R., and Redstone, J. Paxos made live: An engineering perspective. pp. 398–407.

[3] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G., and Abraham, I. Hotstuff: Bft consensus in the lens of blockchain, 2019.