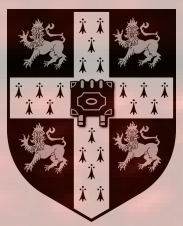
Marc Harvey-Hill

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

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 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 [1]. 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.

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 [3, 4], 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

[5], a cancelled permissioned blockchain-based payments system. The algorithm is relevant because of various performance advantages over other byzantine consensus algorithms such as PBFT [6], SBFT [7], DLS [8], Tendermint [9], and Casper [10].

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 [11].

The main contributions of this dissertation are:

- Providing a reference implementation of HotStuff in OCaml based on a paper by Yin et. al [12].
- 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).
- 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)
- Giving a complete specification and proof of HotStuff (section 3.3), that adapts the pacemaker mechanism which was not sufficiently 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 our changes to integrate the pacemaker with HotStuff without the need for synchronised clocks.

Chapter 2

Preparation

In this chapter we 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

I had some experience using OCaml from the IA Foundations of Computer Science course but had never used it in a project. The IB Distributed Systems course also provided some useful background knowledge, particularly as it briefly covered Raft [13], a non-byzantine consensus algorithm. I had some understanding of byzantine consensus from my own reading into Nakamoto consensus [2] and from developing a wallet application for Ethereum [3, 4]; 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 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 *replicas* (another word for nodes). After several messages are exchanged the log may be committed, meaning that there is consensus on the committed part of the log, and it is now immutable.

HotStuff has the following 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.

Responsiveness[1] — the system is able to make progress as fast as network conditions allow once a non-faulty leader is elected; it does not have to wait for some timeout to elapse.

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

Partially Synchronous — messages sent by one party will always be delivered to another within some bounded amount of time (δ) after global synchronisation time (GST) has been reached.

Authenticated — a message source cannot be spoofed. We assume that all messages are signed, providing authentication.

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. This is the maximum number of byzantine nodes for which consensus can theoretically be reached [14][15].

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 $n - f$ acks[2] before proceeding to the next phase, meaning that at least $f + 1$ of the acks were from honest nodes. The basic HotStuff algorithm has five phases in each view.

The final phase of a consensus algorithm is *decide*. We assume that a *client* sends commands to the nodes, the nodes reach consensus on a log of commands, and then execute them in order; this ensures all nodes will be in the same state. The decide phase commences after the log is committed, the nodes can execute the new commands and respond to the client that the command was successfully committed.

Basic HotStuff is a responsive byzantine consensus algorithm (section 2.2.3). We will start by describing a simpler consensus algorithm that is not responsive or byzantine (section 2.2.1). We then extend this algorithm to handle byzantine threats (section 2.2.2). Finally, we extend this algorithm to be responsive by removing the need for a timeout to progress, arriving at the basic HotStuff algorithm (section 2.2.3).

2.2.1 Non-byzantine consensus

We will start by describing a generic 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 [13] and Paxos [16, 17].

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.

[1] To be precise HotStuff is *optimistically* responsive, we will describe this in section 2.2.3.

[2] N.B. the size of a quorum is different in the non-byzantine case.

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 (λ) to propose. It then broadcasts a *commit* message to the replicas, proposing λ' to the nodes, where λ' is λ 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 has the *safety* property. Since the leader waits to receive a quorum of greater than $\frac{n}{2}$ *new-view* messages, λ is guaranteed to be the longest log that has previously been committed. This is because the the quorum of *new-view* messages must share at least one node with the past quorum of *commit* acks for λ. The new proposal λ' will never conflict with λ, hence the algorithm is safe.

2.2.2 Byzantine consensus

In this section we extend our non-byzantine algorithm (section 2.2.1) to achieve consensus under a byzantine system model. To do this we must first introduce the *quorum certificate* (*QC*), a cryptographic proof that a leader has received a quorum of acks. We then consider the threats posed by byzantine nodes, give an algorithm that solves these problems, and make an argument for safety. Examples of similar algorithms include xyz

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 [18, 19]; in this case the signature from each ack is combined[3]. QCs have a key property that our byzantine algorithm will rely on:

Property 2.2.1. *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 honest node would require at least $2(f + 1) = 2f + 2$ honest nodes, but our system only has $2f + 1$.

Threats

Byzantine nodes introduce two threats that we will deal with in turn. We present ideas for solutions to these threats; it will become clear why these solutions are effective in our argument that the algorithm is safe.

[3]Recall that our messages are signed to provide authenticated delivery.

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.

Algorithm

Modifying the non-byzantine algorithm (section 2.2.1) to include our 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 δ until it receives a *new-view* message from all replicas (we will revisit this in section 2.2.3), and then picks the longest log it received (λ) to pre-propose. It then broadcasts a *prepare* message to the replicas, pre-proposing λ' to the nodes, where λ' is λ optionally extended with the leader's new value. The replicas ensure that λ' 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 λ' to the nodes, and includes a QC of *prepare* acks. The replicas then *lock* on λ' and send a *commit* ack.

decide — same as the non-byzantine algorithm.

Argument for safety

We give an informal inductive argument of the safety of this algorithm based on it solving threats 2.2.2 and 2.2.2. To be safe, we must have that if some log λ is committed in view v, at no point in future will some conflicting log be committed.

Base case — in view v no log that conflicts with λ 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 2.2.1, 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 2.2.1, there must be at least one honest node in the intersection between this QC, and

the QC of *commit* acks λ in view v. This honest replica is locked on λ, 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 λ be committed, so the algorithm is safe.

2.2.3 Optimistic responsiveness

In this section we finally give the basic HotStuff algorithm by extending our generic byzantine consensus algorithm (section 2.2.2) to make it *optimistically responsive*. 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 [20].

The byzantine consensus algorithm is not responsive as it has a timeout in the *prepare* phase. We first describe the problem that means this timeout is needed, then present an algorithm that solves it and make an informal argument that it does not break liveness.

Problem

For the system to have liveness, the leader must wait for δ 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 higher-view log than the other replicas, as in some past view it received the *commit* message, but other replicas did not[4]. When the leader sends the *prepare* message, x will not vote the pre-proposal as it is locked on a higher value, so the leader will not acquire a quorum of acks to make progress; this breaks liveness.

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.

Algorithm

Modifying the byzantine algorithm (section 2.2.2) to include our 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 log from the key with the highest view.*

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

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

The changes ensure that the leader will make progress as it is guaranteed to receive a *key* for a log (λ) if some honest replica is locked on it. This is because for some replica to become locked on λ there must be at least $f + 1$ honest nodes that have a *key* for λ; the leader must hear about λ from one of these honest nodes when it receives a quorum of *new-view* messages.

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.

[could be more concise!!!]

2.3.1 OCaml

I chose OCaml [21] 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 [22] 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 [23] and EIO [24]. I chose Lwt over these libraries due to superior documentation and statbility.

2.3.3 Cap'n Proto

Cap'n Proto [25] 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 [26] 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 [12]. 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, and justify that this project is ethical.

2.5.1 Development methodology

For this project we used an iterative waterfall development methodology. Objectives were chosen in accordance with the timetable set out in the proposal [reference appendix!]. Development then proceeded in cycles of implementation, testing, and analysis of the program trace and timing statements. This approach was particularly useful during the optimisation of our system (section 3.4), which required extensive analysis of the logs and rapid development of different prototypes 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[5] of the consensus state machine code, the coverage report is at *_coverage/index.html*.

The Memtrace library and viewer [27] 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!] [28]

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.

[integrate this with the above to make one paragraph!!!] As mentioned in section 2.5.2, the nature of the project meant that debugging had to be carried out by manual inspection

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

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 analysing the program trace and timing statements, then attempting to test my hypothesis while controlling other variables, and finally implementing a solution.

[CI???]

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 already existing blockchain ecosystem. Such software has many positive applications, but by its nature can facilitate the creation of exploitative markets. Since this software already exists, our contributions will not enable any new forms of unethical markets and products.

Chapter 3

Implementation

In this chapter we describe the architecture of our implementation (section 3.1), conclude our theoretical explanation of HotStuff by discussing the chained algorithm and the pacemaker (section 3.2), present a full specification for HotStuff with a proof of correctness and liveness (section 3.3), describe key optimisations implemented (section 3.4), present our 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

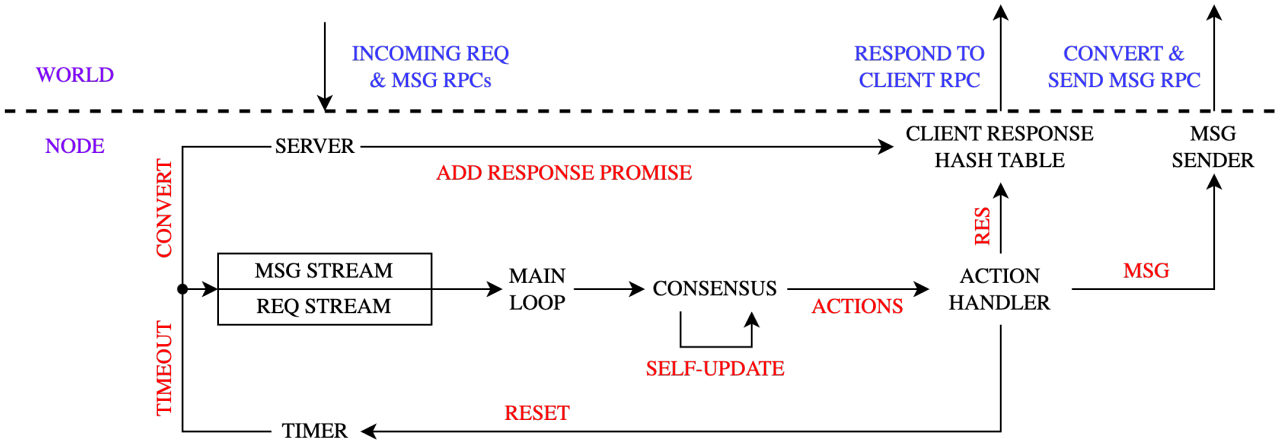


Figure 3.1: Architecture of a node.

In this section we present 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] [29], which was developed by my project supervisor.

We will follow the path of an incoming request or message travelling through the system, as shown in figure 3.1.

[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.

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.

3.2 More HotStuff theory

In this section we conclude our theoretical explanation of HotStuff by discussing chaining, and the pacemaker. As the pacemaker was not sufficiently specified in the original paper, we will draw on other sources to give a full explanation.

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

3.2.1 Chaining

In this section we describe 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 [12].

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. 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 each view is the length of a single phase and a QC can serve in multiple phases concurrently. The only exception to this is the *new-view* phase which is the same as in the basic algorithm.

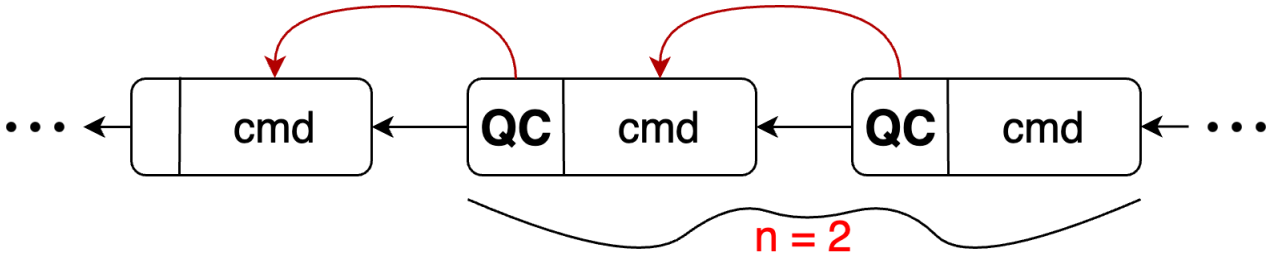


Figure 3.2: Sequence of nodes forming a 2-chain [label parent and justify links!!! fix font].

In each view a chain of nodes is extended, and different phases are carried out on nodes depending on how long a chain they form. For example, figure 3.2, shows a 2-chain, which means a proposal has already been through 2 phases[3]; this is the equivalent of being in the *commit* phase in the basic algorithm.

3.2.2 Pacemaker

In this section we discuss the pacemaker, which is responsible for the liveness of the system; ensuring that it is able to makes progress. The original paper did not sufficiently specify the pacemaker mechanism, so we have synthesised information about pacemakers from different sources (including our own experimentation) to give a full specification of HotStuff in section 3.3. Part of the pacemaker that we will explain 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 [5], which is explained in a talk by Ittai Abraham [30]. We will also discuss how the pacemaker can be integrated with the HotStuff algorithm; we did this through our own modifications based on deadlocks encountered during implementation.

The notion of a pacemaker and the properties it should posses were formalised in the Cogsworth byzantine view synchronisation protocol [31]; we will later prove that our pacemaker has these properties, and that they lead to liveness (section 3.3.2). Pacemakers are related to the concept of a failure detector: a system that facilitates the detection of failed nodes [32, 33]. A pacemaker extends this idea, allowing it to be used in a multi-shot setting.

[3]Not counting the new-view phase, as this can be considered to be the end of the previous view.

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.

Integration

Our 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 are 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 we 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 our chosen design.

3.3 Specification

In this section we give 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). We informally justify some of the changes made from the original algorithm, then give a proof of liveness for our modified algorithm.

This specification implements the consensus module as described in our discussion of the system architecture (section 3.1). This module receives messages and requests from the main loop, outputs actions (sending a message, responding to a client request, or resetting the view timer), and updates its own state.

We present the pseudocode 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 3.4.

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: $n \leftarrow b_{new}.justify.node$
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(), VOTEMSG$_u$(GENERIC, b_{new}, $\perp$))
27: UPDATE(b_{new})
28: **if** not ISNEXTLEADER() **then**
29: ONNEXTSYNCVIEW($curview + 1$)
30: **procedure** ONRECEIVEVOTE(VOTEMSG$_v$(GENERICACK, b, $\perp$))
31: **if** 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: $b_{\text{new}} \leftarrow$ CREATELEAF(b_{leaf}, cmd, qc_{high}, $b_{leaf}.height + 1$)
41: BROADCAST(MSG$_v$(GENERIC, b_{new}, qc_{high}))
42: **return** b_{new}

3.3.1 Changes

We now informally justify some of the key changes made to the original algorithm, with a specific focus on the changes we made to integrate the pacemaker with HotStuff (section 3.2.2).

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)
```

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 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 includes a QC, allowing replicas to catch up if they are in a lower view.

Algorithm 2, line 1 — we 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 $v + 1$

3.3.2 Proofs

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 δ. [For this proof we consider the consensus machine (algorithm 1) and the pacemaker (algorithm 2), to be separate entities.] - relate this to cogsworth synchroniser!!!

[prove liveness with properties!!!]

Theorem 3.3.1 (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 3.3.2 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 δ. 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 δ 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. □

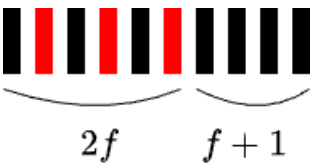


Figure 3.3: Example of round-robin leader allocation for $f = 1$. Red rectangles denote byzantine leaders. [maybe stripe red nodes so they appear in black and white???]

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.*

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 3.3). Hence there will always be at least 2 consecutive honest leaders (the lemma holds trivially for $f = 0$). □

Lemma 3.3.3. *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 [quickly justify that we cannot 'skip ahead' without a qc from a future view]. □

Theorem 3.3.4 (Synchronisation validity)**.** *The pacemaker will only advance the view 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

In this section we give solutions to some of the practical challenges of implementing HotStuff, and describe optimisations we made to improve performance. 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 our 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. We 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

A standard technique to improve the goodput of a consensus algorithm is to *batch* requests, 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.

In this section we discuss the practical challenges of implementing effective batching. 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 we made to make batching effective.

Filtering

One optimisation that we implemented is *filtering* incoming commands to prevent them 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. The effectiveness of this optimisation demonstrated in our 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}
```

We give pseudocode for an implementation of filtering in algorithm 3. We use a set 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 our 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 actually 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

We further improved the effectiveness of batching by limiting the size of each batch. We demonstrate that this approach improves performance in our 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 our 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 Nodes

This section concerns the challenges of designing a suitable type for nodes[4] that can be efficiently serialised by Cap'n Proto and sent over the network. Recall from our discussion of the system architecture (section 3.1) that the internal type for a node must be converted into a Cap'n Proto type in order to be serialised.

Nodes are implemented as OCaml records, and contain the fields *parent*, *cmds*, *height*, and *justify*. The *justify* field contains a QC that contains another node inside it.

Recursive types

One difficulty 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 $b_0.justify.node = b_0$.

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

[4] *Node* is another word for log in this context.

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

In order to reduce memory usage, we replaced 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, 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 equality

One optimisation implemented was 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 through profiling the memory usage of the node equality function using Memtrace.

Node truncation

We implemented node truncation in order to reduce the size of nodes being sent over the network, 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 for our implementation (section 4.2.1). The effectiveness of this optimisation demonstrated in our ablation study (section 4.3.3).

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

In order to truncate the node, our 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 our 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 we use a TCP-style approach. We include 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. 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.

3.5 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.

3.5.1 Load generator

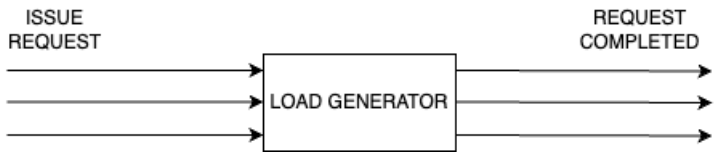


Figure 3.4: Open-loop load generator.

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.

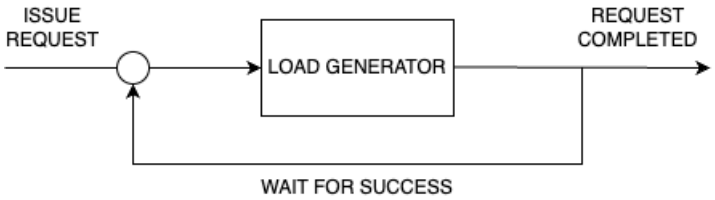


Figure 3.5: Closed-loop load generator.

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 3.4), which means that it dispatches a request every δ 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 for our experiments 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.

Our 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 chose 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 we start the experiment, 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.

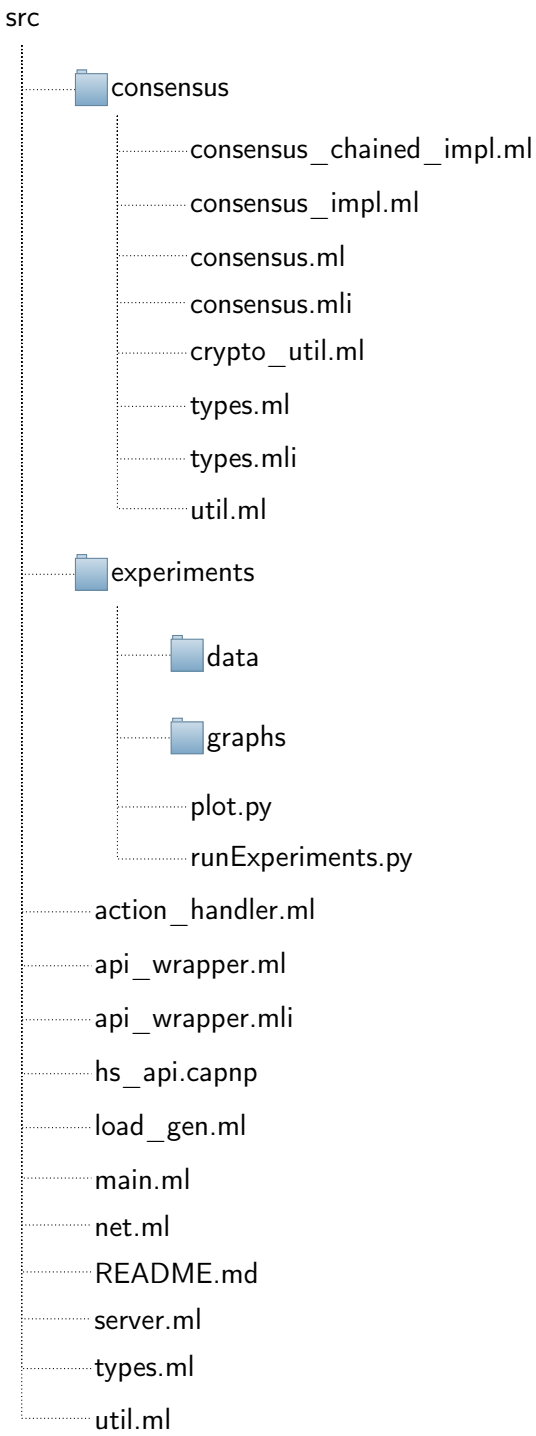
3.5.3 Logging framework

We 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. Our previous approach printed the time taken throughout the execution of the program; since printing takes signficant CPU time 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, storing data, and moving work (such as outputting statistics) to less critical areas of the program.

3.6 Repository Overview



The *src* directory contains files implementing the main components described in the system architecture (section 3.1). This includes the server (*server.ml*), main loop (*main.ml*), action handler *action_handler.ml*, and the message / request sender (*net.ml*). 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 HotStuff algorithms, sharing a common signature (*consensus.mli*). There is also a utility code

that allows the use of the *consensus* module types with cryptography functions (*crypto_ util.ml*).

The *experiments* folder is where the data from running experiments is outputted to, and it 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

In this section we highlight the methods and hardware used in evaluation (section 4.1), benchmark the performance of Cap'n Proto and the Tezos cryptography library (section 4.2), and finally evaluate the performance of our HotStuff implementation (section 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 3.5.1), and were automated using Python scripts (section 3.5.2). 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

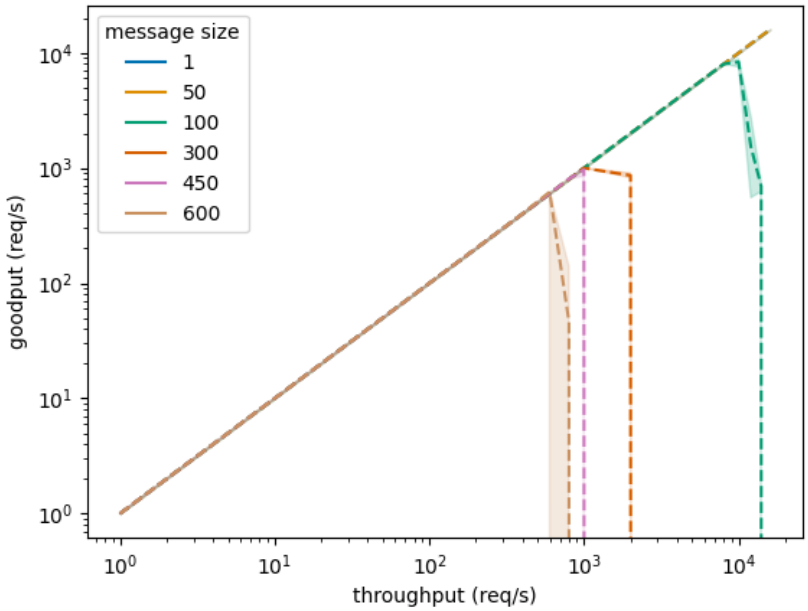


Figure 4.1: Benchmarking of Cap'n Proto server goodput for varying throughputs and message sizes.

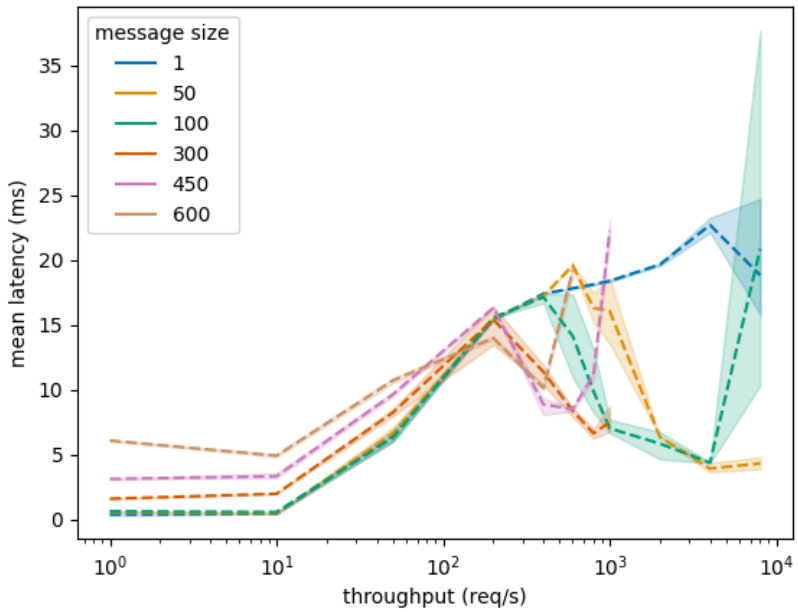


Figure 4.2: Benchmarking of Cap'n Proto server latencies for varying throughputs and message sizes. Discarded result if goodput was not within 5% of target throughput.

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

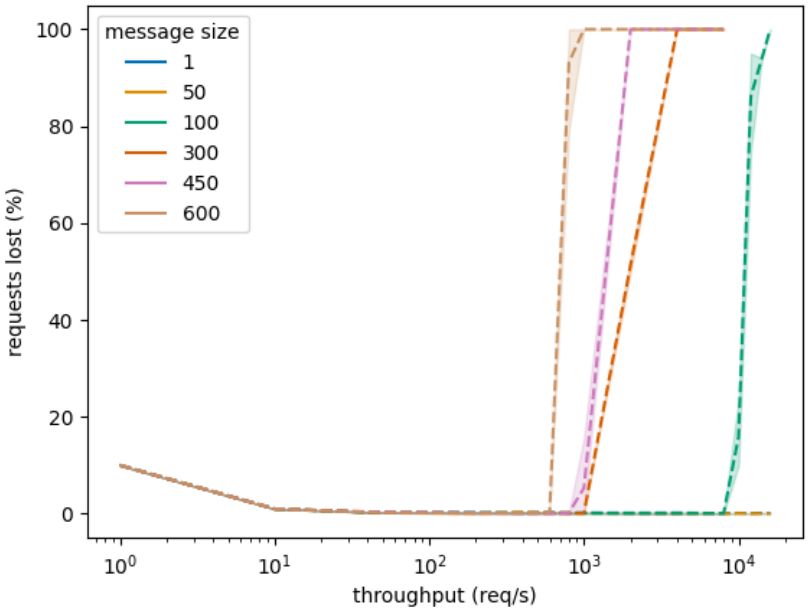


Figure 4.3: Benchmarking of Cap'n Proto server % of failed requests for varying throughputs and message sizes, run for 10s

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 3.4 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 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

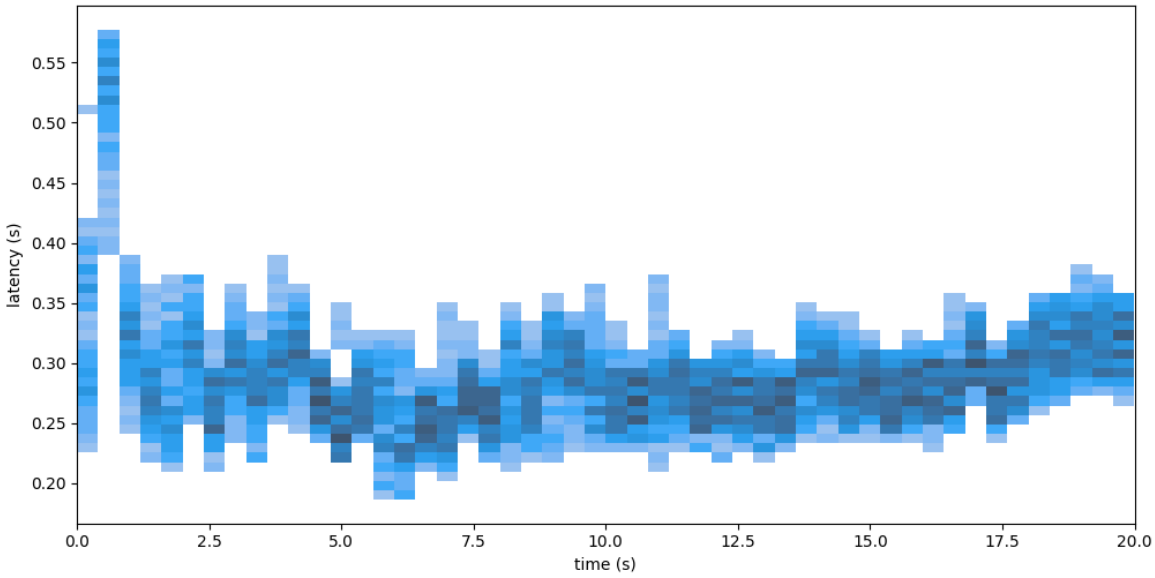


Figure 4.4: Heatmap showing relatively stable latency throughout the course of the experiment. [add parameters!!]

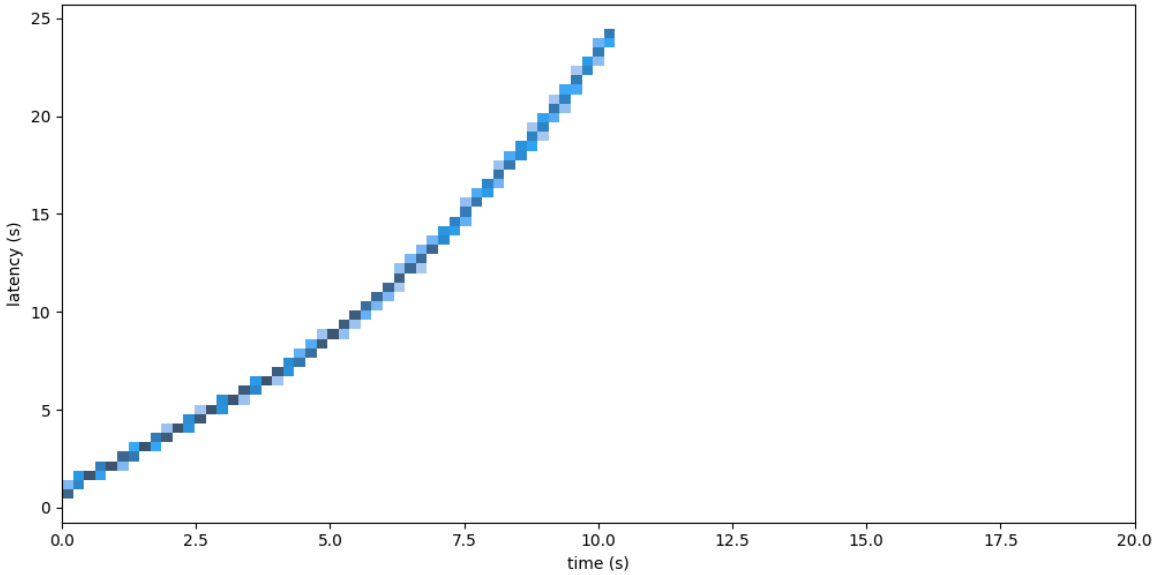


Figure 4.5: Heatmap showing linear growth in latency as the experiment progresses. [add parameters]

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 3.4.1) 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 may observe in a wide area network. [we found...]

In our view-change study (section 3.2.2) we demonstrate that the view-change protocol (section 3.2.2) is effective in ensuring the system progresses once a node has died, albeit with a significant performance penalty.

4.3.1 Batch sizes

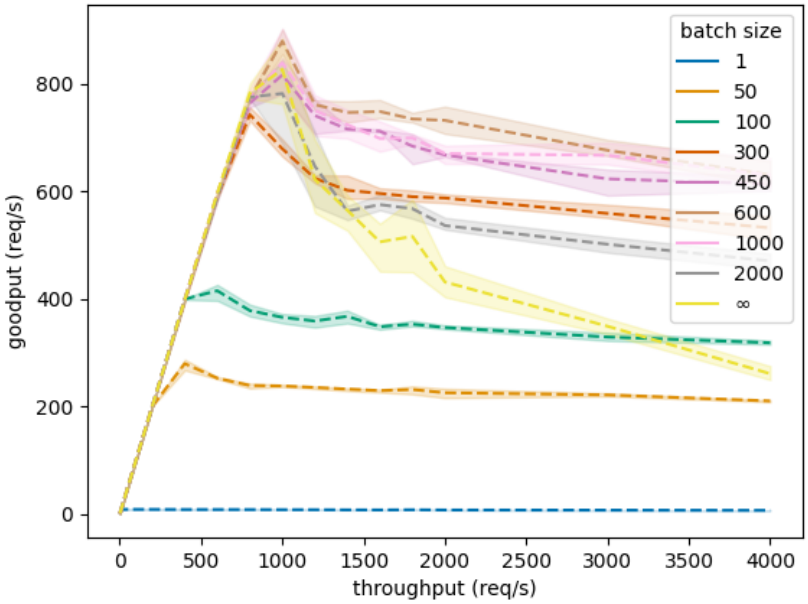


Figure 4.6: Benchmarking of goodput for varying throughputs and batch sizes.

This study compares the performance of the system for varying limits on batch sizes (section 3.4.1). All experiments were run on a network of 4 nodes.

At lower throughputs where the system is not overloaded, throughput grows linearly with goodput (figure 4.6), as the system is able to respond to all incoming requests, 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

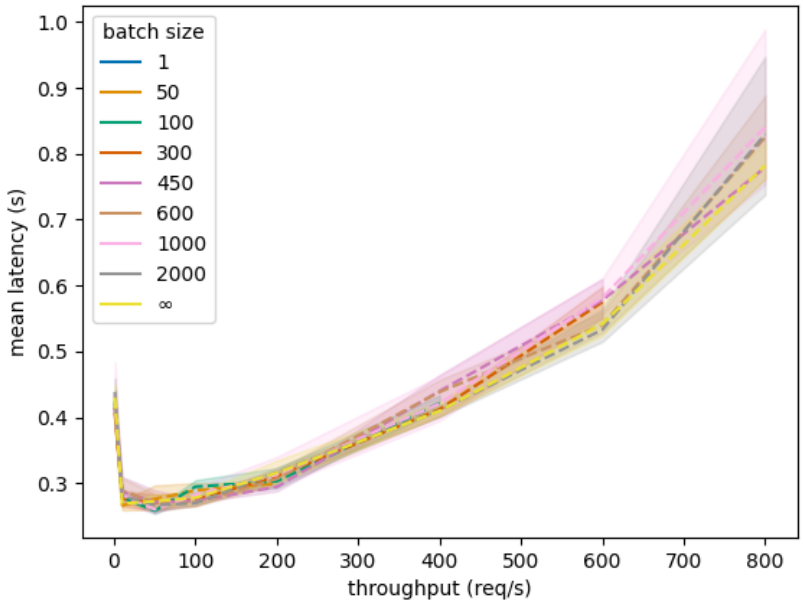


Figure 4.7: Benchmarking of mean latency while varying throughputs and batch sizes. Discarded result if goodput was not within 5% of target throughput.

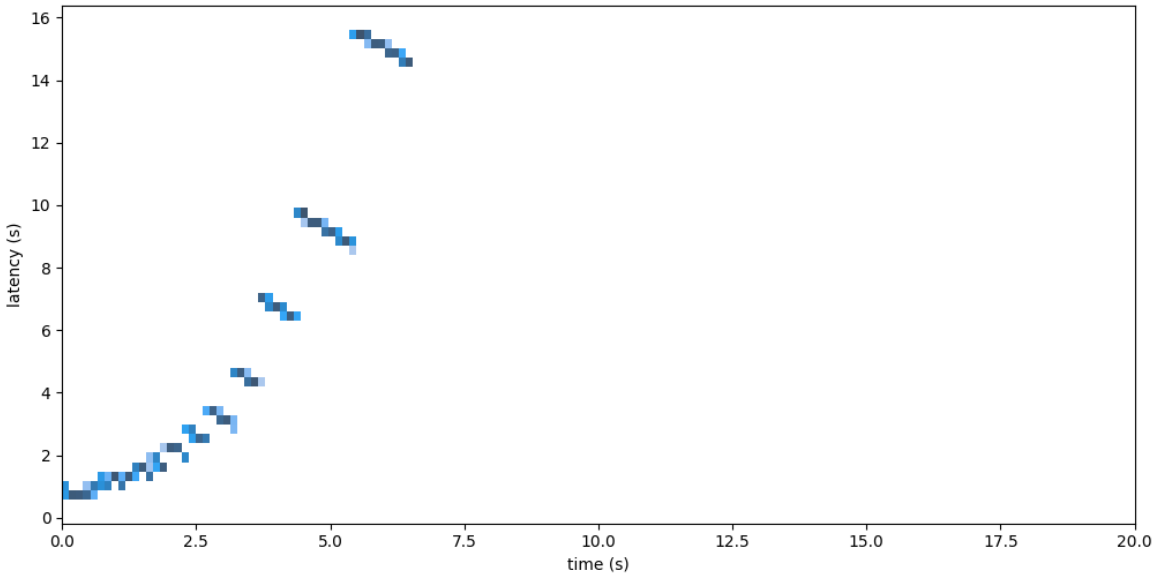


Figure 4.8: Heatmap showing exponential growth in latency as the experiment progresses. [add parameters]

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

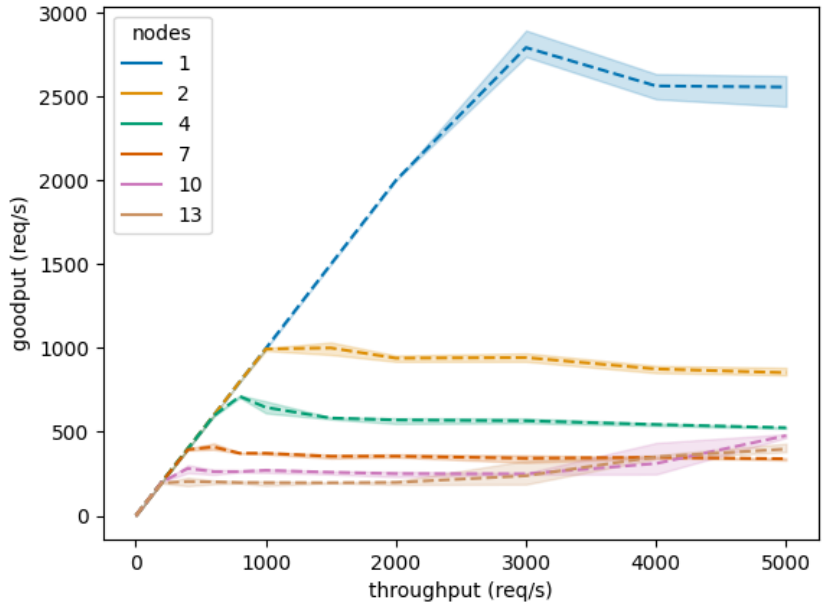


Figure 4.9: Benchmarking of goodput for varying throughputs and node counts.

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.

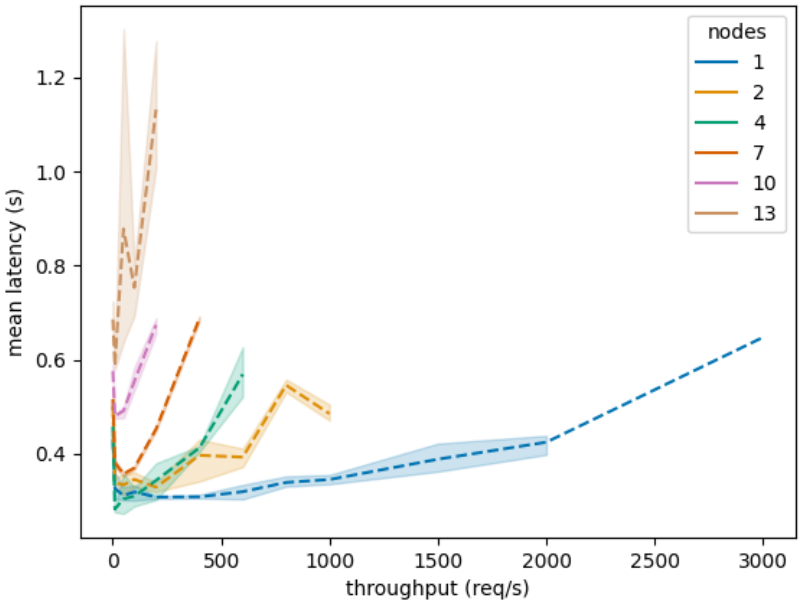


Figure 4.10: Benchmarking of mean latency while varying throughputs and node counts. Discarded result if goodput was not within 5% of target throughput.

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	Chaining	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 3.2.1), node truncation (section 3.4.2), and command filtering (section 3.4.1). We also compare performance with, and without cryptography enabled. 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.]

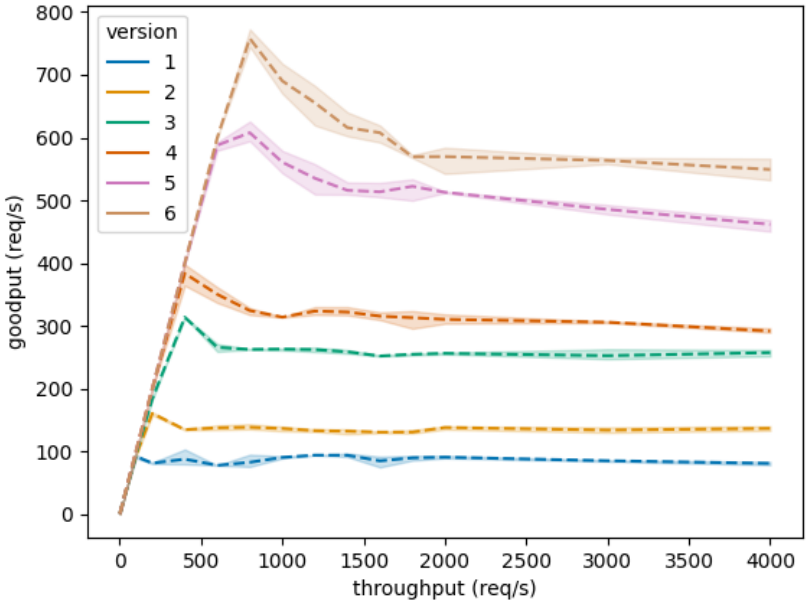


Figure 4.11: Benchmarking of goodput for varying throughputs and implementation versions, run for 10s with 4 nodes unlimited batch size.

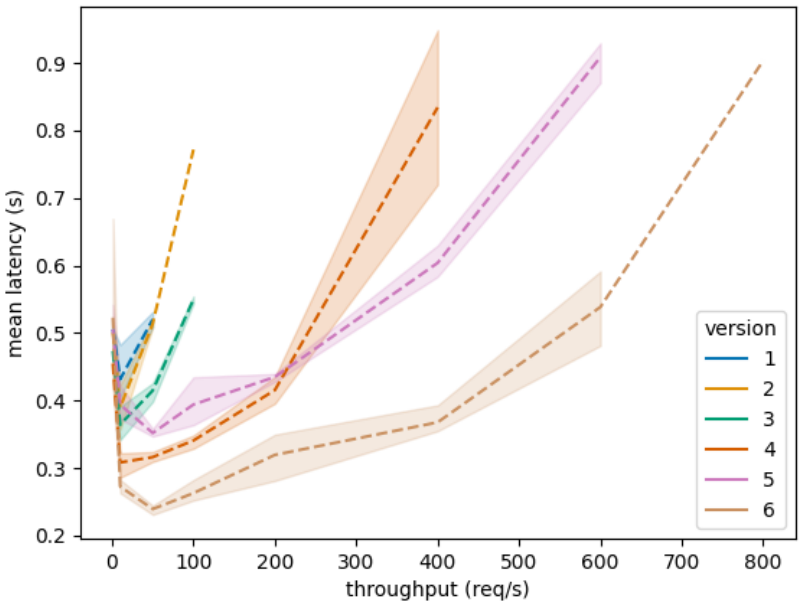


Figure 4.12: Benchmarking of mean latency while varying throughputs and implementation versions, run for 10s with 4 nodes unlimited batch size. Discarded result if goodput was not within 5% of target throughput.

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

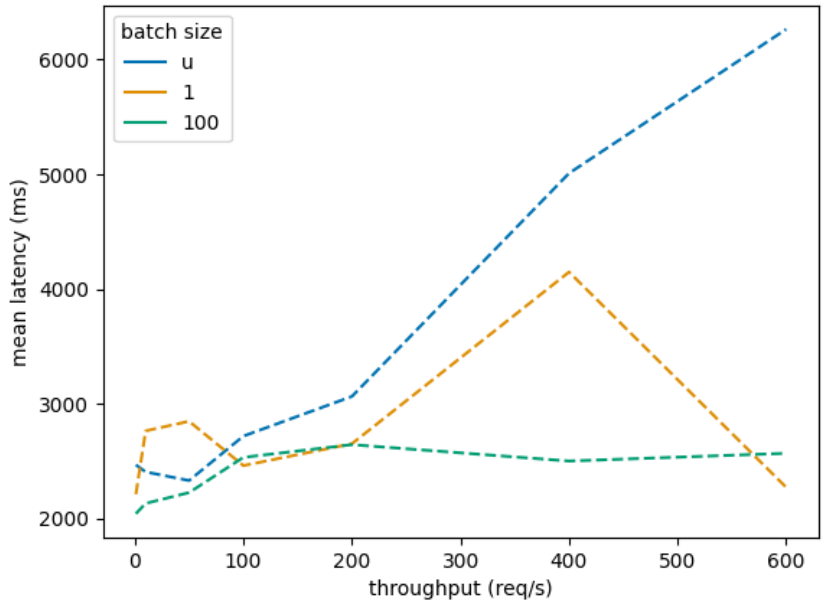


Figure 4.13: Benchmarking of mean latency while varying throughputs and batch sizes, run for 10s with 100ms network delay.

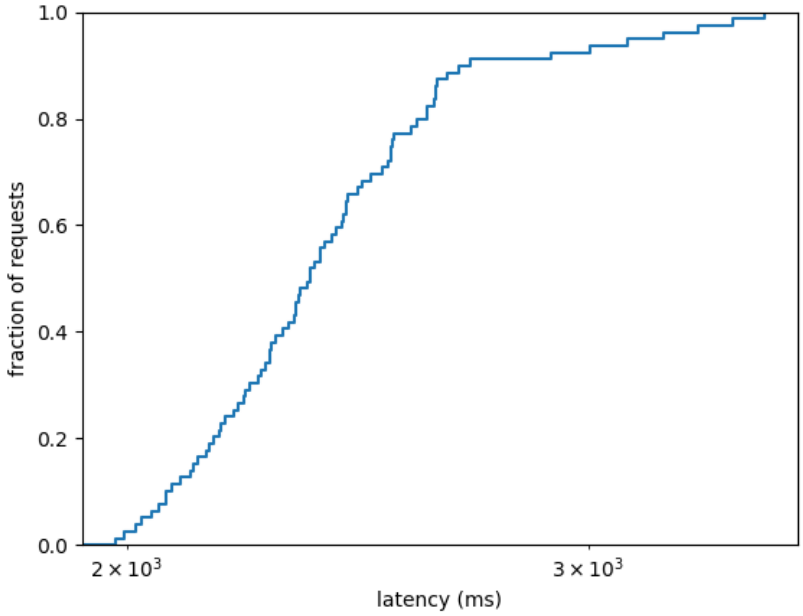


Figure 4.14: Cumulative latency plot for experiment with a throughput of 10req/s and unlimited batch size, run for 10s with 100ms network delay.

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).

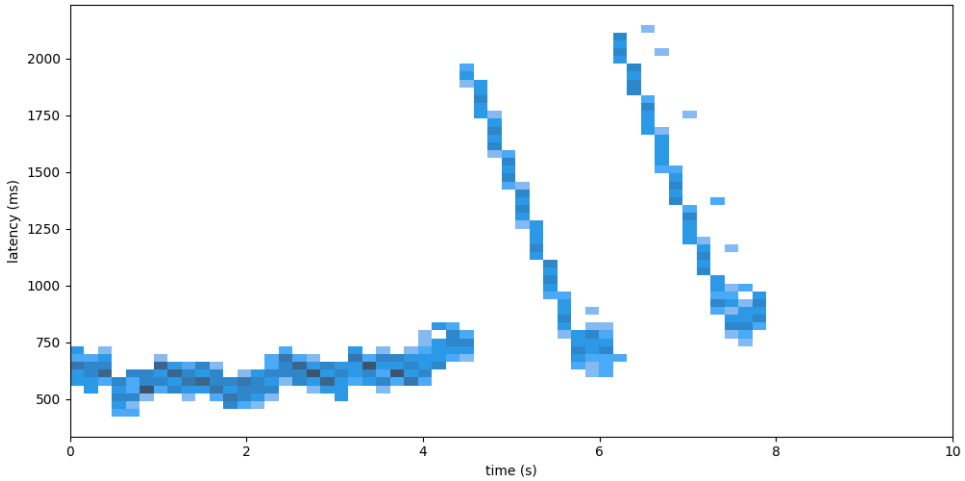


Figure 4.15: Heatmap showing distribution of latencies with a node being killed 5s in. Run for 10s with 7 nodes and a batch size of 100.

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

[consistent tense???] In this project we 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. We 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; we gave a full specification and proved that our pacemaker has desirable properties (section 3.2.2).

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 3.4), 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 [**?**]; 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 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. One 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 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 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

[1]Since our project is implemented purely in OCaml, it could be deployed in a MirageOS unikernel [34]

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] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[2] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Technical report, 2008.

[3] Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. Technical report, 2014.

[4] Gavin Wood. Ethereum: A secure dentralised generalised transaction ledger. Technical report, 2022.

[5] 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.

[6] 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.

[7] 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.

[8] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[9] Jae Kwon. Tendermint : Consensus without Mining. 2014.

[10] Vitalik Buterin and Virgil Griffith. Casper the Friendly Finality Gadget, January 2019.

[11] 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.

[12] 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.

[13] 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.

[14] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[15] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, March 1986.

[16] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[17] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[18] 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.

[19] 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.

[20] 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.

[21] OCaml Programming Language.

[22] Lwt Library. Ocsigen.

[23] Jane Street. Async Library.

[24] MultiCore OCaml. Eio Library.

[25] Cap'n Proto RPC Library.

[26] Tezos Cryptography Library.

[27] Jane Street. Memtrace Memory Profiler & Viewer.

[28] Mininet.

[29] Chris Jensen. OCons Paxos Implementation.

[30] Ittai Abraham. Byzantine fault tolerance, state machine replication and blockchains. SPTDC, St. Petersburg, 2019.

[31] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems*, 1(2), October 2021.

[32] 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.

[33] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[34] MirageOS.