

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]

Contents

Introduction	4
Preparation	6
2.1 Starting point	6
2.2 HotStuff algorithm	6
2.2.1 Problem statement	6
2.2.2 Non-Byzantine case	7
2.2.3 Byzantine case	8
2.2.4 Optimistic responsiveness	9
2.2.5 View changes	11
2.2.6 Chaining	11
2.3 Tools & Libraries	12
2.3.1 OCaml	12
2.3.2 Lwt	13
2.3.3 Cap'n Proto	13
2.3.4 Tezos cryptography	14
2.4 Requirements analysis	15
2.5 Software engineering practices	16

2.5.1	Development methodology	16
2.5.2	Testing & debugging methodology	16
2.5.3	Source code management	17
Implementation		18
3.1	Overview	18
3.2	Pacemaker Specification	19
3.2.1	Correctness Proof	22
3.2.2	Liveness Proof	22
3.3	Deadlocks and performance improvements	22
3.4	Implementing for evaluation	23
3.5	Repository Overview	25
Evaluation		27
Conclusion		28
5.0.1	Future Work	28
Bibliography		29

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 DAOs that have decentralised non-hierarchical governance, internet infrastructure like servers and DNS servers can be decentralised, and any possible algorithm can be run on a decentralised ‘world computer’. The innovative algorithm underlying blockchains is a solution to the byzantine consensus problem, which allows a group of participants to agree on some shared history (such as a transaction ledger), even while some malicious participants try to undermine the process.

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. Permissionless blockchains can be seen as permissioned blockchains 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 blockchain applications within a group or organisation, such as a company, where the participating machines are known in advance; but they can also be used in a permissionless context with the addition of a proof of work / stake mechanism.

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 comparable algorithms like PBFT, DLS, Tendermint, and Casper.

Blockchains generally provide anonymity or pseudonymity; some privacy cryptocurrencies like Monero use zero-knowledge proofs to make transactions anonymous and unlinkable. This has the advantage of protecting the privacy of internet users, and allowing them to evade censorship and surveillance by tyrannical regimes. However this anonymity can also facilitate unethical behaviour such as

money laundering, and the trade of illicit goods like firearms. A potential solution to this problem are verifiable anonymous identities, in which the participants are anonymous in most cases, but identities can be verified when a transaction is called into question (eg. due to a regulatory requirement).

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 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 [4].
- Outlining key practical challenges and considerations of implementation.
- Adapting the pacemaker mechanism presented, giving a full specification that works in asynchronous environments without synchronised clocks
- Outlining how one could implement verifiable anonymous identities [3] using my implementation.

Preparation

2.1 Starting point

I had some experience of 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 were directly useful to implementing HotStuff, but they gave me some wider context of the field.

2.2 HotStuff algorithm

2.2.1 Problem statement

HotStuff is a Byzantine fault-tolerant consensus algorithm. It 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. The main application of HotStuff is in permissioned blockchains. For example, one could create a cryptocurrency by using HotStuff to reach consensus on a log of transactions like "Account X transfers account Y £10". By allowing a large amount of devices to reach consensus on a ledger, one can develop a decentralised payment system. In general blockchains can be applied in situations where there is some central authority (a bank, DNS server, government, etc.) to instead create a distributed and decentralised system that requires less trust from participants.

The protocol can be viewed as a solution 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 the HotStuff protocol the generals can ensure that they all attack together. In contrast to the generals problem, HotStuff is able to agree multiple values instead of just one, so instead of deciding a single value like “Attack at dawn”, HotStuff can agree on a 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.

These conditions 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 GST has been reached and a message source cannot be spoofed. The Byzantine assumption means a maximum of n faulty nodes may be controlled by an adversary that is actively trying to prevent us from correctly reaching consensus, where $n = 3f + 1$ and n is the total number of parties.

2.2.2 Non-Byzantine case

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. Examples of similar algorithms include Raft and multi-shot Paxos. Such an algorithm must ensure that once a value is decided and appended to the log, it cannot be modified or erased. In each view a leader proposes some log which it aims to decide upon with a group of replicas. Our implementation assigns leaders to views using a round robin system.

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, that 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 the 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, when 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.

1. New-view:

- (a) leader \rightarrow replicas: "view = 3"
 - (b) replicas \rightarrow leader: "view = 2, log = ['hello', 'world']", ...
2. Commit:
- (a) leader \rightarrow replicas: "log = ['hello', 'world', '!"]"
 - (b) replicas \rightarrow leader: "ack", ...
3. Decide:
- (a) leader \rightarrow replicas: "decide"
 - (b) replicas: execute log

2.2.3 Byzantine case

In order to extend our algorithm to achieve consensus under a Byzantine threat model we must handle three threats that we will deal with in turn. In order 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'.

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 send in the *prepare* phase, this may also be extended with the leader's new values. Once we receive a quorum of acknowledgements, we begin the *commit* phase. The difference is that this time we include a quorum certificate over the quorum of *prepare* acks, which proves that we pre-proposed the value to at least $n - f$ nodes and received their acks, so we are not proposing one value to some node 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 quorum certificate that they receive during the *commit* phase and will only accept a new *prepare* if it extends from the node stored in the certificate.

3. Threat: A faulty replica sends a the leader a fake log in its new-view message that was never actually proposed. Note that this does not break safety as a pre-proposal for the fake log would not be accepted since the protocol is safe. However, this breaks the liveness property of the protocol as a non-faulty leader could be prevented from making progress by faulty replicas sending fake logs.

Solution: Add to the *new-view* message a qc over *prepare* acks from the original view in which the message was proposed as a proof that it was indeed proposed.

1. New-view:

- (a) leader \rightarrow replicas: "view = 3"
- (b) replicas \rightarrow leader: "log = ['hello', 'world'], qc = (prepare acks from view 2)", ...

2. Prepare:

- (a) leader \rightarrow replicas: "log = ['hello', 'world', '!"]"
- (b) replicas verify the proposal is safe
- (c) replicas \rightarrow leader: "log = ['hello', 'world', '!'] ack", ...

3. Commit (lock):

- (a) leader \rightarrow replicas: "log = ['hello', 'world', '!"]", qc = (prepare acks from previous stage)"
- (b) replicas 'lock' on proposed log
- (c) replicas \rightarrow leader: "ack", ...

4. Decide:

- (a) leader \rightarrow replicas: "decide, qc = (commit acks from previous stage)"
- (b) replicas: execute log

2.2.4 Optimistic responsiveness

Consider again the *prepare* phase. In this phase the leader selects the quorum certificate from the highest view that it hears about from the *new-view* messages. However, it is possible that there is some honest replica that we do not hear from (perhaps their message was lost) that is locked on a higher view proposal than the one that we choose to propose. When this replica receives our pre-proposal, it will reject it as it is locked on a higher view proposal. This means that we could be

prevented from making progress in this view by missing one replica in the *new-view* phase.

This means that our system doesn't have the 'liveness' property, which means that it will make progress under synchronous conditions when a non-faulty leader is elected. It is possible that we repeatedly don't hear from this one honest replica and fail to make progress indefinitely. One solution to this problem is to introduce a timeout Δ that we must wait for before progressing to ensure that we have allowed sufficient time to pass such that we have heard from the honest replica. This solution has the disadvantage that our system is not 'responsive', which means that the system can make progress as fast as network conditions allow when we have a non-faulty leader, and does not depend on Δ .

In order to achieve responsiveness we can modify our algorithm by adding a *pre-commit* phase in between *prepare* and *commit*. This ensures that if some honest replica becomes locked on a value in the *commit* phase, then there are at least $f + 1$ honest nodes that have a 'key' for that value from the *pre-commit* phase. More specifically, they have a 'key-proof' composed of a quorum certificate over *pre-commit* acks. Replicas then send this key-proof with their *new-view* message when a new view begins. The new leader selects the key-proof with the highest view proposal, and sends the key-proof along with their *prepare* message. Even if the leader does not receive a *new-view* message from the replica which is locked on the highest view proposal, they must have received the key to this proposal from some honest replica, so their *prepare* message will make progress.

1. New view:

- (a) leader \rightarrow replicas: "view = 3"
- (b) replicas \rightarrow leader: "qc = (key-proof for view = 2, log = ['hello', 'world'])", ...

2. Prepare:

- (a) leader \rightarrow replicas: "log = ['hello', 'world', '!'], qc = (key-proof for view = 2, log = ['hello', 'world'])"
- (b) replicas \rightarrow leader: "log = ['hello', 'world', '!'] ack", ...

3. Pre-commit (key):

- (a) leader \rightarrow replicas: "qc = (prepare acks / key-proof from previous stage)"
- (b) replicas store qc as a 'key'
- (c) replicas \rightarrow leader: "log = ['hello', 'world', '!'] ack", ...

4. Commit (lock):

- (a) leader \rightarrow replicas: "qc = (pre-commit acks from previous stage)"

- (b) replicas ‘lock’ on proposed log
- (c) replicas \rightarrow leader: “ack”, ...

5. Decide:

- (a) leader \rightarrow replicas: “decide, qc = (commit acks from previous stage)”
- (b) replicas: execute log

2.2.5 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 ***]

2.2.6 Chaining

The unchained algorithm presented goes through three very similar phases in order to commit a proposal, these phases involve 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 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.

*diagram of n-chain ****

The above diagram 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 $b.justify.node$, this 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 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:

- 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

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 chose 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.

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***].

2.3.3 Cap'n Proto

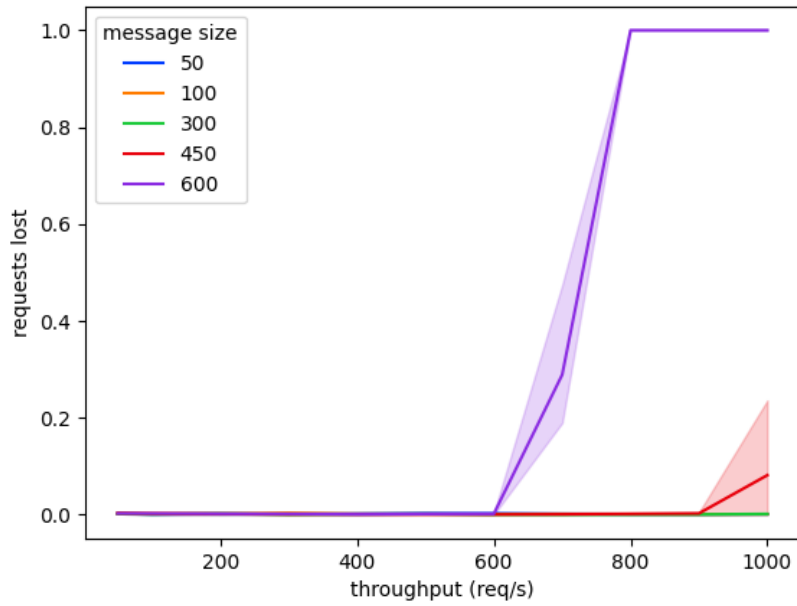
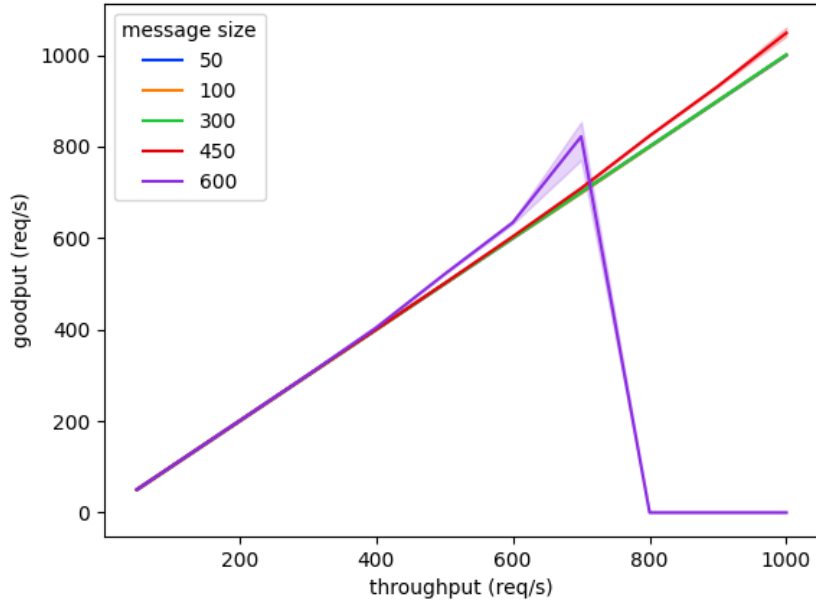
Cap'n Proto is an RPC framework that includes a library for sending and receiving RPCs, and a schema language for designing the format of RPCs that can be sent.

Benchmarks

I benchmarked the message sending functionality of Cap'n Proto. I did this with an open-loop load generator (which is described in 3.4), running each experiment for 10s. I 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.

[maybe add goodput / latency graph once it is in a reasonable state! ***]



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, to aggregate several signatures into a single one, and to check 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

I profiled the important functions of the library with Jane Street’s `Core_bench` module. `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 (μ s)
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

2.4 Requirements analysis

- **Correctness** - The consensus algorithm is implemented as it is described in the paper. This can be established by testing of the program trace for compliance with the algorithm specification [do I need to mention change from proposal here ***].
- **Evaluation** - Analysis of system throughput and latency carried out on a simulated network of 8 replicas. Evaluation will be carried out by testing the program locally, analysing the trace, and testing in an emulator.
- **Improve transaction throughput and reduce latency.** This can be achieved through architectural decisions, tuning the scheduler, and ensuring cryptographic libraries are being used efficiently.
- **Description of how to implement verifiable anonymous identities on top of the HotStuff implementation.**

These requirements are similar to those presented in my proposal (Appendix X***) with a few differences. The first difference is to evaluate on 8 nodes rather than 32. Benchmarking of Cap’n Proto has revealed its limitations when sending large messages. Once batching of requests is implemented the internal messages sent between nodes will be large and could cause a performance bottleneck for the state machine progressing. Because of this it may not be feasible to get reasonable

performance with more nodes, as more nodes result in more internal messages being sent. In practice permissioned blockchains are often run with a small number of nodes, so this limitation may not be important [citation needed *** honeybadger BFT?].

Additionally the extension has been changed from adding support for network reconfiguration to describing how to implement verifiable anonymous identities. This is because this extension seemed like a more interesting direction for the project with more exciting applications.

2.5 Software engineering practices

2.5.1 Development methodology

[should i say i used something like RAD and productivity tracking tools???

2.5.2 Testing & debugging methodology

Unit testing was carried out using ‘expect tests’, which compare the outputted program trace to the correct output. I wrote a testing suite that verifies that the program behaves as specified in the HotStuff paper.

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.

Due to the distributed nature of the program, normal debugging tools and profilers are not useful to 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.

[implement CI???

2.5.3 Source code management

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

Implementation

3.1 Overview

The core implementation of the HotStuff algorithm (which we will give in 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 project¹, 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 (see 2.3.4) for signing messages, aggregating signatures, and checking quorum certificates.

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 3.4). The format of RPCs is specified in a Capn' Proto schema, in their custom markdown language. A received RPC must be decoded, and the Capn' Proto types converted into the internal types of the consensus state machine. Messages and requests are added to separate streams² 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.

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

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

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 Capn' 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 (see algorithm 4 & 5) 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), we will present these in 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.

- CREATELEAF (Algorithm 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. This is very similar to the CREATELEAF function given for the chained version of the protocol in the HotStuff paper; it is unclear as to why this was changed for the unchained version [work out why it changed?].
- ONRECEIVEPROPOSAL (Algorithm 1): The change on line 22 ensures that we only respond to a proposal from our current view, this is important for safety but was not explicitly included in the original pseudocode. The change on line 27 is that we send a NEWVIEW message to the next leader once we receive a proposal. This is in contrast to the original pseudocode where we send a NEWVIEW inside ONNEXTSYNCVIEW on receiving some unspecified interrupt. Finally, once we have received a proposal we can transition into the next view *unless* we are the next leader, in which case we must wait to collect the VOTEMSGs before transitioning.
- ONRECEIVEVOTE (Algorithm 1): The change on line 31 ensures we ignore messages from earlier views, which is important for liveness, and that we are the correct destination for the vote message. Another change we make is dividing V into different sets for messages from different views; this prevents votes

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( $MSG_v(GENERIC, b_{new}, \perp)$ )
22:  if  $m.view = curView$  then
23:    if  $b_{new}.height > vheight \wedge (b_{new} \text{ extends } b_{lock} \vee n.height > b_{lock}.height)$  then
24:       $vheight \leftarrow b_{new}.height$ 
25:      SEND(GETLEADER(), VOTEMSG $_u(GENERIC, b_{new}, \perp)$ )
26:      UPDATE( $b_{new}$ )
27:      SEND(GETNEXTLEADER(), MSG $_u(NEWVIEW, \perp, qc_{high})$ )
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  $\exists (v, \sigma') \in V_{m.view}[b]$  then
33:      return
34:       $V[b] \leftarrow V_{m.view}[b] \cup \{(v, m.partialSig)\}$ 
35:    if  $|V_{m.view}[b]| \geq n - f$  then
36:       $qc \leftarrow QC(\{\sigma | (v', \sigma) \in V_{m.view}[b]\})$ 
37:      UPDATEQCHIGH( $qc$ )
38:      ONNEXTSYNCVIEW( $m.view + 1$ )
39: function ONPROPOSE( $b_{leaf}, cmd, qc_{high}$ )
40:   $b_{new} \leftarrow$  CREATELEAF( $b_{leaf}, cmd, qc_{high}, b_{leaf}.height + 1$ )
41:  BROADCAST( $MSG_v(GENERIC, b_{new}, \perp)$ )
42:  return  $b_{new}$ 
```

Algorithm 2 Modified Pacemaker

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

from different views being used to form a quorum. The change on line 38 means that once a leader has received a quorum of messages, it can transition to the next view (which it starts by sending a proposal in `ONNEXTSYNCVIEW`)

- `GETLEADER` (Algorithm 2): 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): This function previously just sent a `NEWVIEW` message to the next leader. Our modified function updates the `curView`, and resets the timer for the new view. Additionally it calls `ONBEAT` which causes a leader to propose a new value.

- **ONRECEIVECLIENTREQUEST** (Algorithm 2): On receiving a client request we simply add it to a queue of commands waiting to be proposed.
- **ONTIMEOUT** (Algorithm 2): When our current view times out we send a **COMPLAIN** to the next leader, and reset our timer for the next view. This behaviour is explained in 2.2.5.
- **ONRECEIVECOMPLAIN** (Algorithm 2): This function is very similar to **ONRECEIVEVOTE**, except that we are collecting a quorum of **COMPLAIN** messages rather than votes. On achieving a quorum we 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 2.2.5.
- **ONRECEIVENEXTVIEW** (Algorithm 2): 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 Correctness Proof

3.2.2 Liveness Proof

3.3 Deadlocks and performance improvements

Batching & limiting of batch sizes. deduplication in batching Send to all

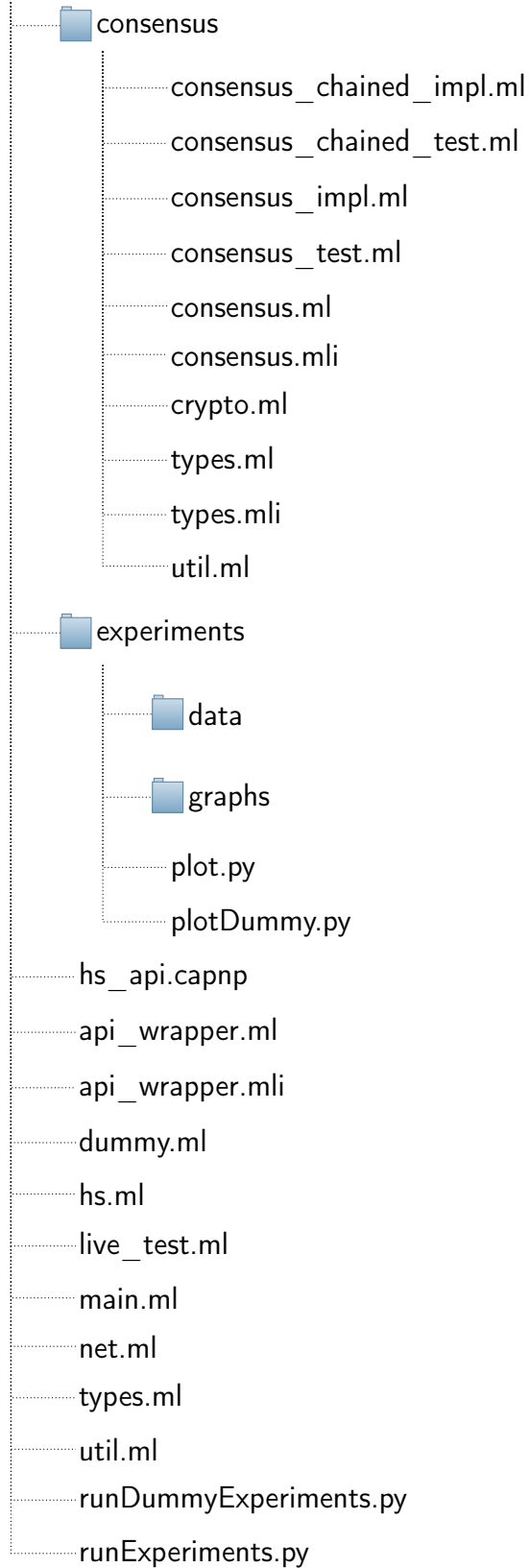
We give cases of deadlock conditions and performance issues / improvements found from debugging traces: 1. print statements (even when not in-between timing commands) can affect the time measured in another operation. Removed prints and printed all results at the end (use static collector function for logging that is passed around the program). 2. \geq used lexicographic ordering, resulted in chains being verified due to first field being increasing until a specific command "20" where the value became lexicographically decreasing 3. Infinitely recursive start node not compatible nice with messaging system 4. Memory usage massively increased due to recursive node functions, replaced with an offset 5. Store a hash in each node and use it to compare nodes rather than checking their entire history 6. Discarding messages from future views results in some leaders not progressing

3.4 Implementing for evaluation

Benchmarking code discuss open loop vs closed loop clients avoid coordinated omission by using open loop clients

3.5 Repository Overview

src



The *src* directory contains the main server loop and the code for interacting with Capn' 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. The *experiments* folder is where the data from running experiments is outputted to, and it contains scripts for plotting graphs.

Evaluation

base on evaluation chapter of papers!

Conclusion

5.0.1 Future Work

Further work would port to using the `async` library which is known to have better performance. It was out of scope to rewrite in `async` or use it in the first place due to poor documentation (although now I could look at the type signatures and understand the documentation). Hopefully reimplementing would give better performance and avoid the bugs of `capnp-rpc`. I would carry out more extensive tests on the message sending capabilities before diving into implementation. I would be more aware beforehand of the whole algorithm (including the pacemaker code) and implement based on the new pseudocode we have presented and proven correct. This would allow for better structuring of the code.

We have presented a potential path for implementing verifiable anonymous identities and reconfiguration using our permissioned blockchain, future work could consist of a practical implementation of this.

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] HARDJONO, T., AND PENTLAND, A. Verifiable anonymous identities and access control in permissioned blockchains, 2019.
- [4] YIN, M., MALKHI, D., REITER, M. K., GUETA, G. G., AND ABRAHAM, I. Hotstuff: Bft consensus in the lens of blockchain, 2019.