High-Performance Byzantine Fault Tolerant Consensus

Master Thesis

Author(s):
Gebauer, Petr

Publication date:
2023-08

Permanent link:
https://doi.org/10.3929/ethz-b-000628543

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 461

Systems Group, Department of Computer Science, ETH Zurich

High-Performance Byzantine Fault Tolerant Consensus

by

Petr Gebauer

Supervised by

Zhenhao He, Prof. Gustavo Alonso

August 2023

D INFK

Abstract

Byzantine fault tolerance is a well-studied strategy for increasing systems' resilience to attacks but there is also renewed interest in it due to its use in blockchains. Due to its high demands on both computational resources and network, it is a target of a number of optimizations on CPUs as well as a topic for hardware acceleration.

This thesis paves the road towards offloading of the Practical Byzantine fault tolerance (PBFT) protocol to FPGAs by presenting a high-performance modular PBFT consensus implementation, which can be used as a platform for future hardware accelerator experiments. We show that compared to moduBFT, a previous implementation exploring the potential for future hardware offloading, we reach multiple times the throughput without an increase in latency.

Our modular approach allows for measuring the performance of different parts of the implementation as well as offloading different parts of the protocol. Moreover, unlike moduBFT, our choice of C as implementation language brings explicit memory management, which we expect to aid in the use of hardware accelerators. We measure the performance under different scenarios, analyze bottlenecks and discuss different offloading strategies and how they can be incorporated into our implementation.

Contents

Chapter 1

Introduction

It is well known that computers can fail or become victims to malicious attacks. In some applications, relying on a single machine not to fail is unacceptable and instead the service needs to be replicated among multiple nodes performing the same task. That way if one fails, the others can still deliver the correct answer. This is known as state-machine replication. A particularly resilient mode of recovering from failures is the Byzantine Fault Tolerance (BFT), which can tolerate not only unavailable nodes but also ones which have succumbed to an attack by a malicious third party.

In the Byzantine fault tolerant state-machine replication setting, there are n servers sharing the same service state and a limited number of them (up to f) can be faulty. The faulty nodes can manifest arbitrary behavior — besides simply becoming unresponsive, they can send incorrect and potentially malicious messages including ones that pretend to be from other servers. The adversary who controls the faulty servers can also have them communicate with each other in order to coordinate their attack.

The service client sends requests over the network and these requests may require a modification of the service state as well as an answer. A Byzantine fault tolerant protocol prescribes the communication among the servers and between servers and clients to correctly update the service state on the working replicas and to make the correct answer known to the client [11].

Besides being used for state-machine replication, Byzantine fault tolerant protocols have been put to use in blockchains. Blockchain systems allow participants to come to a consensus regarding an ordered log of operations [19]. Some of these [35, 48] have been used to implement cryptocurrencies. Some blockchains, such as Bitcoin [35], operate in *permissionless* mode, where the list of participants is not controlled. More recently, we have seen introduction of *permissioned* blockchains [41, 5], in which there is an a priori known set of participants working towards a common goals but without fully trusting each other, enabling the use of BFT [43]. These blockchains can expect to often run in the cloud (e.g. Amazon Managed Blockchain [4] or IBM Blockchain Platform [25, 26]) with the networking advantages this brings.

Multiple BFT protocols have been proposed but one of the most famous is Practical Byzantine Fault Tolerance (PBFT) [11], which can tolerate f faults with $n = 3f + 1$ servers. This protocol is divided into so-called *views*. In each view, one of the servers performs the role of a leader. The client sends its request to the leader, which assigns it an id and broadcasts it to all its peers. This is followed by two phases of all-to-all server communication where in each phase each server confirms the successful completion of the previous phase to all its peers and waits to receive a sufficient number of confirmations from them. Finally, after these two phases, each server executes the request and sends a reply to the client. The client waits for $f + 1$ matching replies before accepting them as correct. This all-to-all server communication for every client request makes the protocol quite communication-intensive. To verify authenticity of messages, cryptographic operations are used. They need to ensure first that the recipient of any message is able to verify the message sender and secondly that the recipient is able to prove the receipt of the message to any third server if necessary. These operations make the protocol computationally intensive too.

In fact, there are voices that BFT implementations will have trouble saturating high-bandwidth networks expected in future permissioned blockchains [43]. This leaves room for optimizations on CPU as well as potentially additional performance gains through offloading some operations to specialized hardware devices, such as FPGAs. There has recently been a surge in the use of FPGAs for different computations, including parts of web search on intermediate [40] and large [12] scale, performing data queries closer to their storage [28], machine learning applications, such as convolutional [49, 8] and fully-connected [51] neural networks, networking stack [23, 18] or compression and encryption [13]

However, most existing BFT implementations use high-level languages, such as Go [43, 45] or Java [7], which poses a challenge for using hardware accelerators. For example, both Go and Java manage memory through the use of a garbage collector, taking explicit memory management, whic is typically required by hardware accelerators, away from the user. Other implementations, like hotstuff [50] use a high-level of abstraction provided by external networking libraries, which have a large codebase. Offloading the functionality of these libraries would be difficult due to their size.

This work presents OBFT, a high-performance modular PBFT implementation written in C suitable for future offloading to hardware. OBFT[1] allows servers to reach a consensus regarding total ordering of client requests. This consensus could be used for state-machine replication of any underlying service.

The text is organized as follows. We first survey related work in chapter 2. Chapter 3 describes our implementation and optimizations used. It contains the design of our networking and cryptographic buffers and thread synchronization schemes. The last part of this chapter, section 3.4, describes our approach to handling some of the possible corner cases. Our evaluation is presented in chapter 4. First, we show performance of two previous implementations, followed by our results in section 4.3. We show that compared to our main reference implementation, we reach multiple times larger throughput at a comparable latency. Next, we evaluate the effect of different circumstances on our performance and in section 4.3.4 we analyze bottlenecks present in our implementation. In the next chapter, we propose a number of methods of improving our performance further. Of particular interest is section 5.2, which describes, how a hardware accelerator could be integrated into our implementation. We have performed a part of this implementation but more effort is required to make it usable.

[1]The name OBFT mainly refers to the suitability for offloading.

Chapter 2

Related work

2.1 Background

In the original Byzantine fault tolerance problem statement and early solutions [33, 39], the setup was formulated as the problem of Byzantine generals. A number of byzantine generals are observing an enemy city and deciding whether to attack or retreat. An attack requires a sufficient number of generals to join in order to be successful. Each general makes a judgement based on the situation, votes either for attack or retreat and communicates its decision to other generals. However, some generals can be traitors who communicate a different decision to different colleagues. The goal is for all honest generals to decide on the same course of action. Furthermore, this action should make sense — if the enemy position can be easily taken, the traitors shouldn't be able to convince their honest colleagues to retreat and vice versa.

BFT provides stronger security guarantees compared to Crash fault tolerance (represented by protocols, such as Paxos [32] and Raft [37]). In the Crash fault tolerant model, faulty nodes simply become unresponsive. However, Byzantine fault tolerance protocols can handle faulty consensus parties generating messages, including potentially ones carefully designed to confuse other nodes.

Later publications in this area, such as Practical Byzantine fault tolerance [11] often work with state-machine replication. The replicas store a state of a common underlying service and receive client requests. They reach a consensus regarding the requests' content and their order and execute these requests in this agreed order. Since the publication of PBFT, other protocols aiming at improving performance have been published. We describe some of these publications in section 2.5.

More recently, the use of BFT in blockchains has been considered in academic work [9, 10, 20] and even seen practise [47, 19]. Byzantine fault tolerance can be used to derive proof-of-stake protocols [14], which are much less computationally expensive than proof-of-work protocols (e.g. Bitcoin [35]). This renewed interest raises the importance of optimized BFT implementations.

Many BFT protocols including Practical Byzantine fault tolerance are network and computationally intensive, which makes them suitable for hardware offloading experiments. Previous work has shown that networking protocols [23], cryptography [13] and even some crash fault tolerant protocols [27, 3] can be successfully offloaded to hardware, giving hope for offloading of BFT protocols.

2.2 Practical Byzantine fault tolerance

This section describes the PBFT [11] protocol in more detail. The protocol requires $n = 3f + 1$ replicas to tolerate f faults and is divided into so-called views. In a single view, one of the peers performs the role of a leader, also known as the primary, and it receives all client requests. Upon receiving a request, the leader assigns it a numeric id and broadcasts it to its peers along with an accompanying tuple in the form $\langle \text{PRE-PREPARE}, v, n, d\rangle_{\sigma_p}$. Here v is the view number, n is the assigned message id, d is a digest (a hash) of the request and σ_p is a signature of the tuple by the leader. The signature strategies will be discussed later. In this text, we use the terms *server*, *peer*, and *replica* interchangeably and we refer to the non-leader nodes as *followers* or *backups.*

When a replica gets such a pre-prepare message, it verifies the view number and signature and checks the message id falls within an acceptable range. If the message is accepted, the replica starts broadcasting to other peers (including the primary) a prepare message in the form $\langle \text{PREPARE}, v, n, d, i\rangle_{\sigma_i}$ where i is the replica id. The prepare messages are also checked for valid signature, view number and whether message number falls within the acceptable range. If a replica (including the leader) receives $2f$ valid prepare messages, it enters the commit phase, in which it broadcasts $\langle \text{COMMIT}, n, d, i\rangle_{\sigma_i}$. When a peer receives $2f + 1$ of these commit messages, it executes the request and sends a reply to the client. If the client receives $f + 1$ matching replies, it knows they are correct.

Pre-prepare, prepare and commit messages are kept in a log on each replica until the corresponding request has been executed by $f + 1$ non-faulty peers and the replica can prove this fact to other peers if necessary. To avoid the cost of computing these proofs after each request, the paper introduces the notion of checkpoints. Checkpoints are generated periodically, e.g. once every 1000 requests. To produce a checkpoint, a replica broadcasts a message $\langle \text{CHECKPOINT}, n, d, i\rangle_{\sigma_i}$ where n is the sequence number of the last request contained in the checkpoint and d is a digest of the state of the service at the time of the checkpoint. When a replica has received $2f + 1$ correct checkpoint messages, they can be used as a proof of correctness of that checkpoint. Such a checkpoint is called *stable* and all messages preceding it can be removed from the log.

This garbage collection after creating stable checkpoints is used for defining the acceptable range of request ids mentioned earlier. Each node will only accept messages with request ids above its last stable checkpoint but no more than k checkpoint intervals after this checkpoint, where k is a constant.

2.2.1 View changes

The leader can also be faulty just like any other replica. In that case, it needs to be replaced by changing the view. This process is started by timeouts. When a client is unable to obtain $f + 1$ matching replies within a specified time limit, it will broadcast its request to all replicas (we call this action a *resubmit*). A peer is said to be waiting on a request if it received this request but hasn't been able to execute it. Peers do not wait indefinitely, instead when their waiting time exceeds a threshold, they start a two-phase communication to perform the view change.

If a peer i has been waiting on a request for too long in a view v, it starts a view change to view $v + 1$ meaning that leadership will be passed from current primary with id p to node $p + 1$. The peer stops accepting messages (other than checkpoint, view-change and new-view ones) and broadcasts a tuple $\langle \text{VIEW-CHANGE}, v+1, n, \mathcal{C}, \mathcal{P}, i\rangle_{\sigma_i}$ where n is the message id of the last checkpoint known to i and $\mathcal{C}$ is the set of $2f + 1$ checkpoint messages proving its correctness. $\mathcal{P}$ is a system of sets $\mathcal{P}_m$ for each request with id $m > n$ that finished the prepare phase on i. Each set $\mathcal{P}_m$ consists of messages proving the completion of this phase — it contains the pre-prepare message and $2f$ prepare messages for request m.

When the leader $p + 1$ of view $v + 1$ receives $2f$ valid view-change messages, it starts broadcasting $\langle \text{NEW-VIEW}, v + 1, \mathcal{V}, \mathcal{O}\rangle_{\sigma_{p+1}}$, where $\mathcal{V}$ consists of the received view-change messages and $\mathcal{O}$ is a set of pre-prepare messages computed as follows. The leader computes numbers *min-s* and *max-s* where *min-s* is the number of the latest stable checkpoint in $\mathcal{V}$ and *max-s* is the highers request id present among prepare messages in $\mathcal{V}$. Then it creates a new pre-prepare message in view $v + 1$ for every id n between *min-s* and *max-s*. If there is a set $\mathcal{P}$ in the messages from $\mathcal{V}$ containing the id n, the primary generates $\langle \text{PRE-PREPARE}, v + 1, n, d\rangle_{\sigma_{p+1}}$ where the digest d is taken from the message in $\mathcal{P}$. If there is no such set, the leader will use a placeholder d^{null}, which represents a null request. A null request is processed by the protocol in the same way as any other but its executed operation is no-op. After the broadcast, the leader appends messages from $\mathcal{O}$ to its log and if *min-s* is greater than its last stable checkpoint, it also inserts the proof of stability of checkpoint *min-s*. Afterwards, it enters view $v + 1$ and starts accepting messages for this view.

A backup checks a new-view message's signature, its view number and the set $\mathcal{O}$. The verification of $\mathcal{O}$ is done by performing a similar calculation as the leader did when creating its new-view message.

To defend against the new leader potentially also being faulty, followers have a way of aborting the view-change to $v + 1$ and instead start changing view to $v + 2$. When a follower receives $2f + 1$ view-change messages for the same view $v + 1$, it starts a view-change timer. When when the timer has been running for some baseline time T and the replica hasn't received a new-view message, it broadcasts a view-change message for $v + 2$ and this time waits for $2T$. This process can be repeated until the view change succeeds.

2.2.2 Cryptographic operations

Since the setup of byzantine fault tolerance allows the faulty replicas to manifest arbitrary malicious behavior, messages need to be signed to verify their authenticity. In principle, this could be achieved by each replica signing the messages it sends with its private key and verifying incoming messages with the sender's public key. However, using this for all messages would be overly costly. Therefore, the authors present an optimised version, where these private key signatures are only used for the rarely sent view-change and new-view messages. Other messages use so-called *message authentication codes*, MACs, which can be computed much faster.

Each node (replica or client) shares a 16-byte secret *session key* with each replica (each pair of nodes shares a different session key). A MAC is computed by concatenating the message with this session key and calculating a hash of the result and finally truncating the hash to the 10 least significant bytes. The original paper proposes using the MD5 hash but more modern implementations, such as ModuBFT moved on to SHA256. The same session key is used for sending messages from a node A to a node B and for sending from B to A.

The main downside of MACs is the inability of the receiving node to prove the receipt of the message to a third party based only on the MAC. To alleviate this problem, messages are signed with so-called *authenticators*. An authenticator is a vector of MACs with one entry for each replica. Thus if a server A sends a message to B and B can verify the authenticity of the message by looking up its entry in the authenticator. Furthermore, if B later forwards the message to C, then C is can also verify it by looking up its entry in the authenticator. This means that when sending a message, a number of MACs need to be calculated and this number grows with the replica count. However the authors argue that the lower MAC computation time is worth the need to calculate multiple MACs for any reasonable replication factor that could be expected for BFT.

2.3 ModuBFT

ModuBFT [43] is an experimental framework implementing PBFT intended for exploring the tradeoffs present when running BFT in permissioned blockchains.

Expecting these blockchains to be interconnected by fast network and to contain hardware accelerators, the authors focus on performance under these conditions. They run experiments on another implementation to demonstrate the effect that a technique called *batching* has on throughput as well as the downside of increasing latency. In this technique, a number of messages is aggregated and signed with a single signature, reducing the work needed for creating and verifying signatures. The authors explain that they do not use batching in their own implementation to keep the latency of the protocol low (representative of the underlying network latency).

The paper then argues that without the use of very large batches, high-bandwidth networks can hardly be saturated unless hardware accelerators can be effectively leveraged. It presents experiments where the availability of hardware accelerators is simulated by using less expensive (albeit less safe) signature operations and the speedup can be observed. The authors also experiment with using MACs for inter-server or all communication.

2.3.1 Implementation

ModuBFT is implemented in Go and relies heavily on goroutines and Go channels to obtain pipeline parallelism. Goroutines resemble lightweight threads and can be executed on a different CPU cores. A Go channel is a thread-safe bounded FIFO buffer for objects of a certain type, which allows enqueue as well as dequeue operations from multiple goroutines simultaneously.

The moduBFT pipeline consists of stages, each of which is executed by one or more goroutines. Every goroutine only executes one stage, it gets work from the previous stage through a Go channel and passes the result to the next stage through another channel. We will now describe these stages.

We will start by discussing the receiving part. The description is supplemented by figure 2.1. On each node, every open connection has a goroutine associated with it, which reads messages from the connection, unmarshals them and passes them to an output channel, which is common to all connections. Messages from

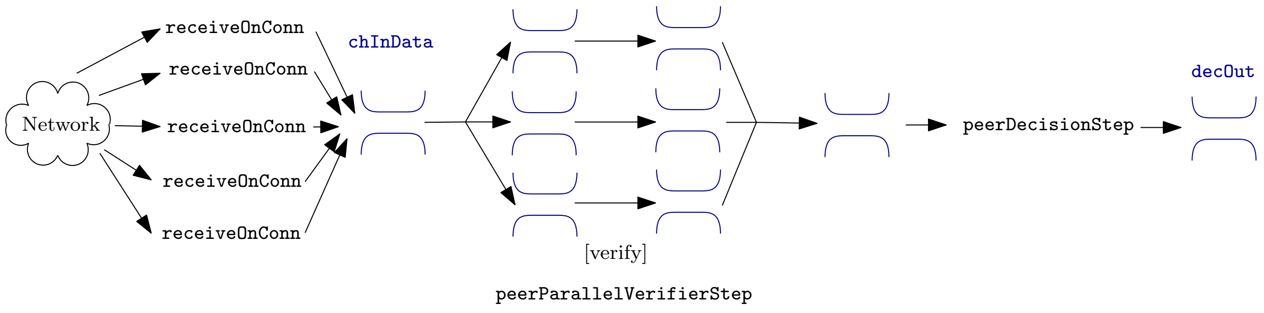


Figure 2.1: ModuBFT receiving path (channels depicted in blue)

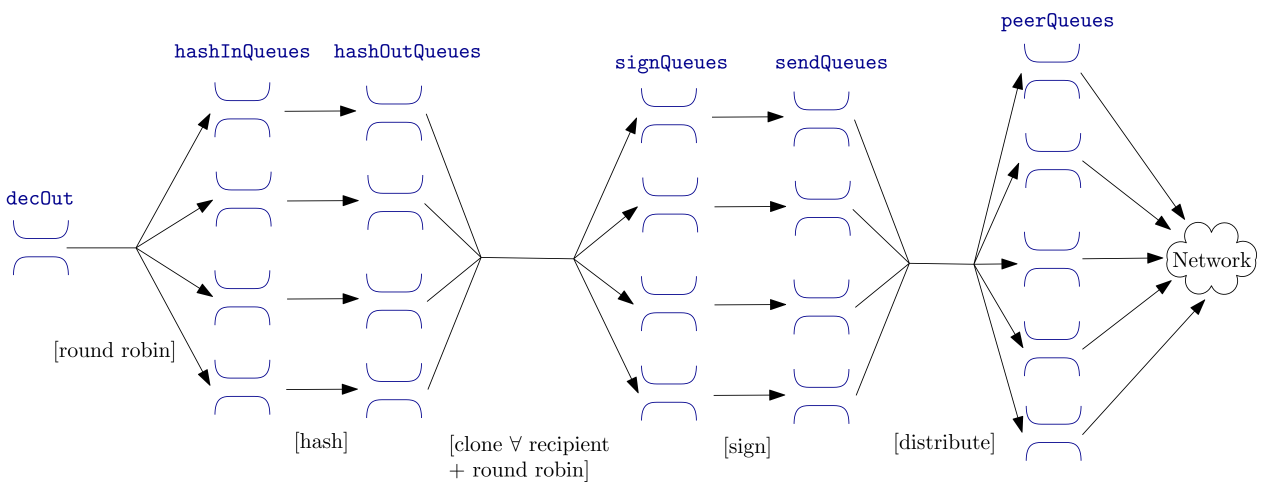


Figure 2.2: ModuBFT sending path

this channel are distributed among a number of goroutines and verified in parallel before being aggregated in a common output channel. The next stage is not parallelized and it implements the state-machine of the protocol; making decisions such as when there are enough commit messages for a particular request to be executed and a reply sent. Any messages this stage wants to send are passed through a channel, `decOut`, to the other part of the program for hashing, signing and sending.

The sending part of moduBFT, depicted in figure 2.2, works as follows. There is a certain number, k, of goroutines used for hashing messages in parallel and the same number for signing them. There is a single goroutine that reads messages from the `decOut` channel and distributes them in a round robin fashion among the hashing goroutines. Each hashing goroutine has its own input and output channels and works with message objects that can still have multiple recipients. The hashing goroutines are followed by a single goroutine that takes messages from the hash output channels and copies each one r ways where r is the number of intended recipients. It marks each copy with the single intended recipient and distributes them in a round robin fashion among the k signing goroutines, which sign them in parallel. Then the messages are picked up by another goroutine, which services all k signing output channels and places the messages in the appropriate sending queue based on their recipient (each recipient has one queue). Finally, each sending queue has its own goroutine, which marshals the outgoing messages and actually performs the send.

2.4 Hotstuff

Hotstuff [50] is an implementation of its own protocol, which enjoys linear authenticator complexity (the number of authenticators being sent) in the number of servers while still allowing progress at the pace determined by actual network latency, rather than an assumed worst-case latency (this is called *responsiveness*). The tradeoff for this combination is somewhat increased latency due to the addition of another phase besides prepare and commit.

Hotstuff uses threshold signatures, in which there is a single public key held by all replicas and each replica holds its unique private key. A single private key allows creating a partial signature and a set of at least $2f + 1$ different partial signatures can be used to create a (complete) signature, which can be verified by the public key. This allows exchanging the all-to-all communication pattern for one where the leader broadcasts a message, receives answers with partial signatures and combines them into a signature, which it uses when it broadcasts the next phase message. This way, the leader sends and receives $\Theta(n)$ messages, each with one authenticator, and the incoming authenticators are combined into one. Thus, not only the number of messages but also the number of authenticators is $\Theta(n)$ per phase, which given a fixed number of phases implies a linear authenticator complexity.

Furthermore, hotstuff streamlines view changes. In the regular PBFT protocol, a view change[1] involves $\Theta(n)$ replicas each sending a view change message to the new leader and this message contains a set $\mathcal{C}$ of $2f + 1 = \Theta(n)$ messages proving the correctness of the last stable checkpoint known to that replica. This means sending $\Theta(n^2)$ message authenticators in total. Next, the new leader has to convince replicas that it really did receive $2f + 1$ new-view messages by sending the set received messages to all replicas. Since each of the view-change messages contains $\Theta(n)$ authenticators, there are $\Theta(n^2)$ authenticators in each new-view message. And as the new leader sends out $\Theta(n)$ new-view messages, it needs to send $\Theta(n^3)$ authenticators, bringing the total number of authenticators sent per view change up to $\Theta(n^3)$. Hotstuff manages to decrease this complexity to $\mathcal{O}(n)$.

This enables hotstuff to change the leader very frequently, even with every request, which in turn allows for a technique the authors call *chaining*. When the leader of view v performs the prepare phase for some request r_1, it passes the responsibility of performing the next phase (pre-commit) to the leader of $v + 1$. However that leader instead starts a prepare phase for a new request r_2. The votes collected during this prepare phase also serve as votes for the pre-commit phase of the previous request r_1. Next, the view passes to $v + 2$ and the votes collected for the new request r_3 serve as prepare votes for r_3, pre-commit votes for r_2 and commit votes for r_1 at the same time.

In its comparison to BFTSMaRt [7], the paper reaches 2–3 times larger throughput.

2.5 Other BFT publications

There is a number of other publications on this topic and comparing to all of them is outside the scope of this thesis. However, we will briefly describe some in this section.

MirBFT The authors MirBFT [45] use the observation that in leader-based protocols, the single-leader bandwidth often becomes the bottleneck and allow for a number of parallel leaders. To ensure these leaders do not make duplicate proposals, thus wasting computational resources, they choose a scheme of partitioning requests among the servers. To prevent a faulty node from censoring incoming requests, this partitioning changes with time. This approach can be applied to the PBFT protocols [45] as well as other leader-based protocols (e.g. hotstuff) [46].

DQBFT Another publication aiming at improving the scalability of Byzantine fault tolerance is DQBFT [6]. The authors separate the total ordering of requests and their distribution among the replicas. The ordering remains centralized to guarantee consistent total ordering, while the distribution can be decentralized.

BFTSMaRt This implementation [7] of PBFT focuses on robustness as well as throughput and modularity. The emphasis is to introduce a complete implementation of the protocol, rather than a greatly simplified research prototype. The authors state that for this reason, they made certain tradeoffs of performance for robustness, including for example implementing the protocol in java, rather than C/C++. They also designed BFTSMaRt to be a library, supporting state-machine replication with an arbitrary underlying service.

[1]Refer to section 2.2.1 for detailed description of PBFT view changes

Zyzzyva Zyzzyva [31] increases performance in the normal case through speculation. The primary assigns request their ids and forwards them like in PBFT. Followers optimistically execute requests as soon as they learn about them and reply to clients. If the client receives matching replies from all $3f + 1$ replicas, it can consider them correct. If some of the peers are slow or faulty, the client can receive between $2f + 1$ and $3f$ matching replies. In this case, the protocol reverts to a two-phase variant. If the primary is faulty, the followers can temporarily become inconsistent. The client can thus receive less than $2f + 1$ matching replies, in which case it triggers a view-change. However, a safety violation has been found in the original protocol. [1]

SBFT This publication focuses on scalability (high number of peers) and performance with geographic replication. The authors employ some of the optimizations from Zyzzyva while avoiding its vulnerability. First, they make the observation that with Zyzzyva's communication pattern, there is no all-to-all communication as messages are collected by the client, which sends out a message to everyone if needed. This reduces the message complexity from quadratic to linear. SBFT [21] implements a similar collector pattern but messages are collected by a round-robin-assigned replica instead of the client. SBFT also employs a speculative fast path. Other optimizations include adding more servers than required for keeping the safety property in order to allow the speculative fast path even if a small number of nodes is faulty.

Bosco In a situation when there is no contention present in the systems — meaning that no two clients submit their requests at the same time, improvements to performance can be made. Bosco [44] takes this approach and introduces a protocol with a single step under these conditions, while still guaranteeing safety if contention occurs. However, it requires $5f + 1$ servers to tolerate f failures. The paper also states and proves lower bounds on server count required for safety with a single consensus round.

Good case latency The work by Abraham et al. [2] focuses the partial problem of performing a broadcast of the request and introduces the term *good-case latency*, used implicitly in many previous applications. Intuitively, this is the latency needed when the machine performing the broadcast is honest. They present a protocol, which optimizes this latency by reducing the number of steps by one compared to PBFT at the expense of needing $5f - 1$ peers to tolerate f failures. This per count requirement is an improvement over a previous implementation, FaB [34], which required $5f + 1$ peers in this scenario. Abraham et al. also prove a lower bound of $5f - 1$ peers. Their algorithm was later improved by decreasing the number of messages sent, which resulted in lower latency [24].

2.6 Crash fault tolerance acceleration

Crash fault tolerance, which is a weaker scheme for recovering from failure than Byzantine fault tolerance, has previously been successfully accelerated with FPGAs. In this section, we briefly describe these publications.

Consensus in a box The work by István et al. [27] uses FPGAs to perform crash fault tolerant consensus resulting in high and stable performance. It aims to reduce the overhead of the consensus protocol sufficiently to remove it from the critical path, while also improving energy consumption. The authors base their system on the atomic broadcast algorithm used in a coordination service named Zookeeper [29] and implement it in hardware. They use a version their previous work on TCP/IP stack in hardware [42] for networking, tailored to the use-case of atomic broadcast in a data center. As part of the evaluation, the resulting system was used to implement a key-value store to test its performance with a realistic application running on the FPGA. The reported overall results form a significant improvement over traditional solutions.

Waverunner This publication [3] implements a crash fault tolerant state-machine replication service based on the Raft [37] protocol. The common performance-critical part is offloaded to hardware, while other, more complex but rarely used routines remain in software. This hybrid approach reportedly reduces development effort without compromising performance. The authors state that compared to the "Consensus in a box" implementation, their system is also simpler and easier to maintain, while likely reaching similar performance.

Chapter 3

Implementation

We have written OBFT in C to enable fine-grained control and explicit memory management in order to aid future offloading. For this reason, we also avoided high-level networking libraries and instead implemented our networking using lower level means, such as the Linux `send()` system call. The explicit memory management allowed us to optimize away dynamic memory allocation on the critical path. Instead, we use bounded circular buffers and during normal operations, memory is allocated once and then keeps being reused. Another design principle we put to use was multithreading. Although the core of the protocol, which maintains the protocol state, is inherently sequential, other, more computationally demanding parts can be parallelized. This includes cryptographic operations and networking. Finally, we aimed for a modular program, meaning we separated the execution of different tasks making it easier to benchmark these parts separately. Besides the benefit to benchmarking, such an approach helps with the integration of hardware accelerators as it allows for offloading a single module without affecting other ones.

The implementation is divided into three modules — networking, state-machine (which is responsible for implementing the protocol logic — e.g. counting prepare messages and deciding when to enter the commit phase) and the cryptographic module. For easier understanding, we provide a design overview in figure 3.1. We chose this modular approach to allow for offloading parts of the protocol to hardware. For example, to offload networking (copying of the message for each recipient and the network protocol implementation), only the networking module would have to be changed. To further offload cryptography, the cryptographic module can be completely bypassed.[1]

Each module has its own set of buffers, where the messages are moved for for the duration of their processing by this module. Incoming bytes are placed into one of the input buffers based on their sender and message boundaries identified by the networking module. The module then aggregates messages from the multiple buffers into one message stream and hands the messages to the cryptographic module.

The cryptographic module moves messages into its single verification buffer, verifies then an passes them on to the state-machine module. This module consumes received messages and creates new messages as appropriate in response. New messages are moved to a dedicated cryptographic buffer for signing and then to the networking buffers. A message intended for a single recipient is placed into the corresponding buffer. Messages intended for all servers are copied by the networking module into all appropriate buffers. Since we keep the content of such messages identical, we can — unlike moduBFT — sign them only once and use this single signature for all recipients.

[1]We offer a more detailed proposal of hardware offloading strategies in section 5.2

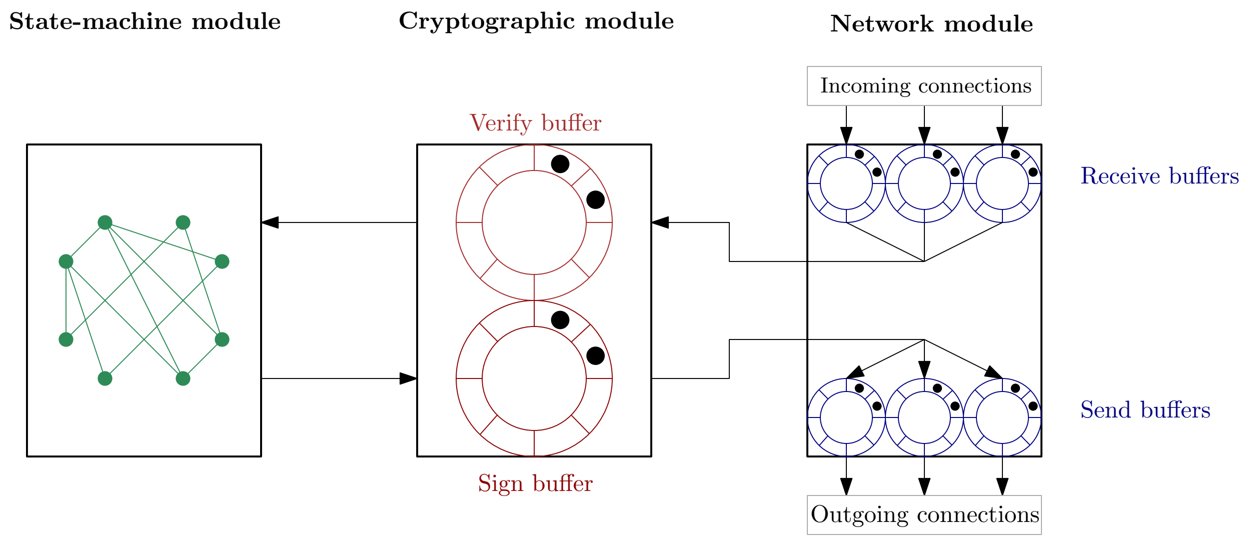


Figure 3.1: Overview of our design

3.1 Message structure

This section describes how messages are layed-out in memory and parsed from the byte stream coming over the network. The layout is depicted by figure 3.2. The message starts with a header containing the following entries:

- `type` — Type of the message, for example `PRE-PREP`, `COMMIT` or `CKPT`
- `fromid` — ID of the node sending the message
- `cliid` — ID of the client who sent the corresponding request. This is used by followers to identify where to send their answers to.
- `id` — ID of the request the message belongs to. In case of checkpoints, this is set to the id of last executed request.
- `view`
- `timestamp` — Clients mark their requests with timestamps to identify them as they are not aware of the id the leader will assign to it.
- `recp` — This field contains the recipient in case there is only one, or a special value, -1, to denote the message is intended for all servers. The use of this special value for message with multiple recipients allows us to keep this field identical in all copies being sent out.
- `len` — Length of the entire message in bytes. It includes the payload and signature.

The header is followed by the digest of the request (or, in case of checkpoint messages, the digest of the service state). The header and the digest have a fixed structure, common to all messages, which allowed us to represent this structure by a C type, `struct msg`. The struct is serialized by mere memcpy. The header entries are all of the same type and the digest is represented by an array of `unsigned char`. For this reason, the danger of misinterpretation caused by any inserted padding in the struct is minimal. The message payload immediately follows the struct. It is treated as an opaque blob of bytes. The last part of the message is the signature — either a public key signature represented as a single monolithic byte segment, or an authenticator.

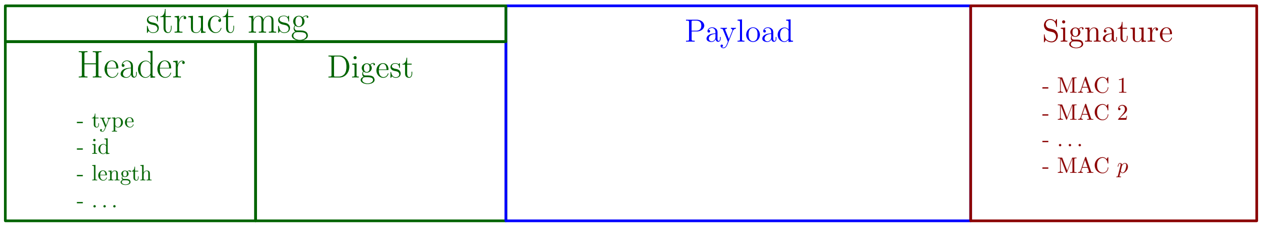


Figure 3.2: Message layout in memory

Parsing We will now describe how a received message can be parsed. The header of each message has the same structure making its entries immediately available for reading. Since the size of `struct msg` is constant, the payload segment start is also known. The receiver is also able to find out the signature length based on its knowledge of the signature structure, which we will describe in section 3.3.1. By subtracting the size of `struct msg` and the signature size from the message length, the node can compute the length of the payload.

When the receiver knows the start of payload and its length, it can compute the signature start and start parsing the signature.

3.2 Networking

In this section, we describe the steps taken to efficiently send and receive messages over the network. We have each node open a socket of type `AF_STREAM` and bind it to its designated port. Connections are initialized using calls to `connect()` and `accept()` and to actually send and receive messages, we use the `send()` and `read()` system calls on these connections.

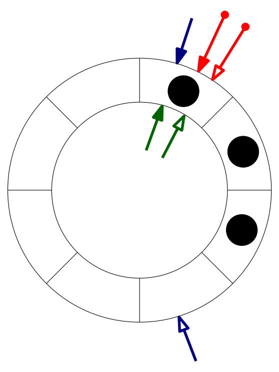
However, since system calls introduce significant overhead, we gather outgoing messages on each connection into a connection-specific outgoing buffer and send multiple messages at once. This allows the `read()` call to receive multiple messages with a single context switch and place then in an incoming buffer, from which they can be read one by one or in groups as desired.

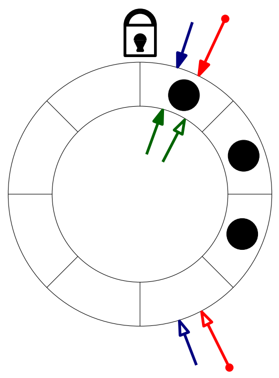
The protocol features an all-to-all communication pattern where every node sends e.g. a prepare message for a given request to all its peers making it common to send the same message on multiple connections at once. To parallelize this, we have a dedicated *sending* thread for each connection. Likewise, each connection has a dedicated *receiving* thread. We use the term *I/O threads* to refer jointly to sending and receiving threads. We will now describe the buffer design and communication between the I/O threads and the thread extracting messages from the input buffers or enqueueing them in the output ones. The buffer design and an example of its operation are depicted in figure 3.3.

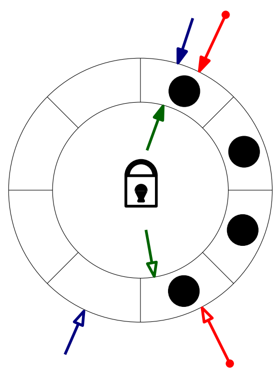
3.2.1 Output buffers

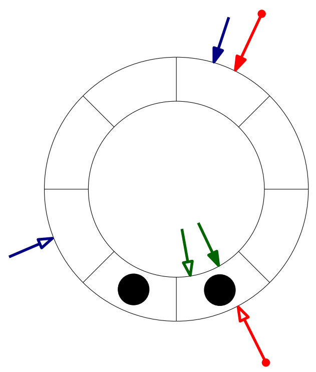
Every output buffer is a circular buffer with a `start` index pointing to the first valid byte inside it and an `end` index pointing one byte behind the last valid byte. The buffer has a fixed size. If `start == end`, the buffer is empty, if `(end + 1) % size == start`, the buffer is full. These indices are shared between the sending thread and the thread enqueueing messages and they are protected by a lock. Initially, this lock was buffer-specific but later, we abandoned this approach in favor of using one lock for all output buffers as this improved performance. To avoid having to obtain the lock every time a message is enqueued, the enqueueing thread has its own pair of start and end indices, `local_start` and `local_end`. When enqueueing a message, it checks if there is space in the buffer by comparing `local_start` to `local_end`, copies the bytes into the buffer and increases `local_end`.

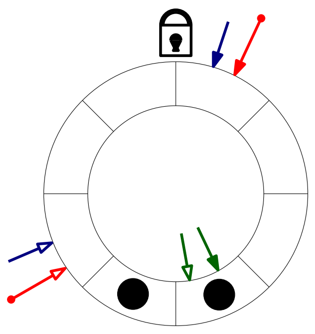
Thus, `local_end` is the most up to date end pointer and the shared `end` has to be updated based on its value. When `end` is not fully up to date, it is less than `shared_end` showing less bytes in the buffer than there really are. Therefore, it is a conservative estimate for the sending thread who can read it, send all bytes between `start` and `end` and update the value of `start` to indicates the bytes are no longer in the buffer. This means that `start` is more up-to-date than `local_start`, which can be smaller. This again creates a


(a) Bytes are enqueued in the buffer and the enqueueing thread advances its end index.


(b) The enqueueing thread obtains the lock and updates the shared end.


(c) The sending thread obtains the lock and updates its end. Meanwhile, the enqueueing thread is free to enqueue more bytes in parallel.


(d) The sending threads performs the send and updates its start (in parallel with more bytes being enqueued).


(e) The enqueueing thread obtains the lock first and updates the shared end.

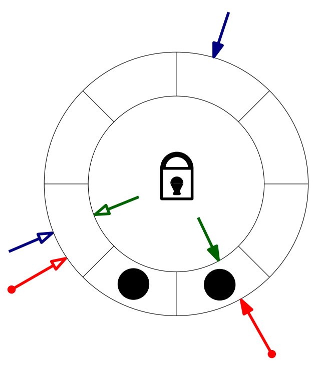

(f) The sending thread obtains the lock, updates the shared start based on its own and its own end using the shared one.

Figure 3.3: Networking buffer design demonstrated on send buffers. Blue arrows on the outside represent indices used by the enqueueing thread, the larger red arrows represent indices shared between the threads and the green arrows inside the circle represent the values of start and end indices that the sending thread is aware of. This last pair of indices is kept in local variables. Arrows with filled tips are start indices, those with hollow tips are end indices. The buffer is protected by a lock. A lock drawn on the inside or outside of the ring means the lock is held by the enqueueing or sending thread respectively. Black circles represent the actual bytes placed in the buffer.

conservative estimate for the enqueueing thread because if it sees a smaller value in `local_start` than what is in `start`, it will assume there is less space in the buffer to put its messages.

When the enqueueing thread compares `local_start` to `local_end` and finds that based on their values a significant portion of the buffer appears to be full, it obtains the buffer lock, updates the value of `end` based on its `local_end` and the value of its `local_start` based on the value of `start`. Then it releases the lock. At some point, the sending thread will acquire the lock, read the value of end, release the lock and start sending the bytes. After the send is finished, it will reacquire the lock and update the value of `start`. To avoid having to get the lock twice for every `send()` call, the sending thread updates the value of `start` when it is reading the value of `end` — it obtains the lock, writes `start` based on the previous `send()`, reads `end`, releases the lock and calls `send()`.

When the sending threads has nothing to send, it needs to wait for the enqueuing thread to place more bytes in the buffer. It needs to keep the lock released during this waiting to allow the enqueueing thread to update the value of `end`. However the sending thread needs to hold the lock whenever it reads `end` to avoid a data race. Rather than unlocking, waiting for some duration and locking to check, we added a condition variable to each output buffer that the sending thread can wait on until the enqueueing thread puts enough bytes in the buffer and signals the variable. Likewise when the enqueueing thread sees that the buffer is full (even after updating its `local_end`), it waits on the same condition variable to be signaled by the sending thread after creating some space in the buffer.

This way, as long as there is some space in the buffer, the enqueueing thread will be able to place a new message in it and as long there are enough bytes in the buffer to justify a system call, the sending thread will be able to keep sending them. Moreover, the common enqueue operation is inexpensive because it doesn't require any synchronization operations unless a significant number of bytes has piled up. What constitutes as a significant number of bytes is configurable. It is always $\frac{1}{n}$ of the buffer but n can be specified using an environment variable (as well as the buffer size itself).

Lastly, although the buffer is logically a ring, its physical layout is a continuous segment of memory with the start and end indices being calculated modulo its size. Therefore, if the valid bytes cross the boundary — e.g. if there are 20 bytes in the buffer and `end == size - 10` and `start == 10` — two `send()` calls would normally be required. To avoid this (and lower the number of system calls needed), the sending thread has its local buffer that it can copy these bytes into and so that they occupy a continuous segment of memory and can be sent by one system call.

3.2.2 Input buffers

The principle of input buffers it to a large extent complementary to the output buffers. Each input buffer is also a circular one with shared `start` and `end` indices and `local_start` and `local_end` for the thread that pops messages from the buffer. In this case, the reading thread puts bytes in the buffer and updates the shared `end`, which is thus the most up-to-date end index (and `local_end` its conservative estimate). On the other hand, the popping thread updates its `local_start`, which is the most up to date, and `start` is its conservative estimate.

There is a lock protecting the shared `start` and `end` indices, which the receiving thread obtains after each `read()` call to update the shared `start` before continuing to the next read. Just like with output buffers, this lock is shared by all input buffers. As long as there's space in the buffer, the receiving thread can do multiple reads without any action by the popping thread. The popping thread consults its `local_start` and `local_end` to find out whether there are any bytes in the buffer. If necessary, it updates its `local_end` using the shared `end`, writing a more up-to-date value in the shared `start` as it does so. It will also update the shared `start` if it is significantly behind its `local_start` to make the available space visible to the receiving thread and enable it to receive bytes before the buffer gets empty.

The difference here is that the popping thread doesn't know which connection the incoming bytes will appear on and has to iterate through the incoming buffer until it finds one which is not empty. And if all the buffers are empty, it cannot wait on a condition variable of any particular buffer because no messages may ever come over that connection. To enable this waiting, we introduced a semaphore, which is shared by all incoming buffers and effectively counts the number of finished `read()` calls. When one of the receiving threads comes out of the `read()` call and updates its buffer's `end`, it also posts this semaphore. When the popping thread finds all its buffers empty even after updating all its `local_end` variables, it waits on this semaphore before iterating through the buffers again.

Finally, just like sending threads, receiving threads avoid having to make two system calls in cases where the available space is divided into two non-contiguous segments of memory. To do this, twice the buffer size is actually allocated and only the first half corresponds to the circular buffer. The bytes returned by the `read()` system call can then overflow into the second half and be copied into the appropriate position at the start of the first half of the allocated segment.

3.2.3 Networking deadlock

We have at times observed the system getting stuck and clients timing out for a reason we call *networking deadlock*. We analyzed the state of the system by having the client crash upon timeout, aborting all other processes. That way, we could examine core dumps of the servers.

The state we observed was that a node A was trying to send messages to a node B. This is done by placing this message in a buffer, where it is taken over by a dedicated thread, which performs the `send()` system call. We saw that this buffer was full (forcing the enqueueing thread to wait) and the sending thread was inside the system call. On node B, the situation was similar — it was trying to send a message to node A but it was still in the system call and the buffer was full.

We will now describe our explanation of where the problem likely lay. Node A was trying to send messages to node B. Initially their receipt was confirmed as per the TCP protocol but as these messages kept coming without `recv()` being called, eventually they couldn't be received. This meant that the `send()` call could not finish because in our setup, it guarantees messages are not lost. That caused the outgoing buffer to get filled and the thread enqueueing messages into this buffer to wait for more space. Thus, node A could only continue operation if node B received the messages sent to it. However, node B could not receive those messages because it was in the same situation trying to send messages to node A.

In our experimental setup, there is a known bound R such that there can be at most R outstanding requests in the system at the same time. Here *outstanding* means the client sent the request and has not received $f + 1$ matching replies yet. Therefore, it would appear this can be resolved by increasing the buffer size so that it can accommodate all messages corresponding to R requests. However, we have observed this fix not working, possibly due to the following problem. When A is processing a group of R requests and does not receive any prepare or commit messages from B but receives them from $2f + 1$ other nodes, it will still reply to the client without waiting for B (after all, B could be faulty). Other nodes may do the same, delivering $f + 1$ matching replies to the client and making it submit another group of R messages. If messages from B are delayed sufficiently, this second group can also be processed without A receiving any messages from B, leading to a third group of R requests being submitted. This can repeat many times before messages from B to A finally arrive, filling A's buffer. If messages from B to A got delayed as well, B's buffer could also become filled causing the deadlock. Therefore, no fixed buffer size may be sufficient in the presence of large network delays.

This may be a principal problem. If A is trying to send messages to B and B does not react to them in any way, the requests have to be kept or dropped. If this continues indefinitely, not all messages can be kept because of memory constraints. It is known that coming to a consensus with no assumptions on the network delays is impossible [17] and for this reason, the authors of PBFT [11] make an assumption about $delay(t)$, the time it takes for a message sent at time t to be delivered if the sender keeps retransmitting it. The assumption is that $delay(t)$ does not grow faster than t indefinitely. However, this still allows the delay to grow to a value that does not allow for retransmitting all undelivered messages because they do not fit into memory.

Hotstuff [50] makes uses a partially synchronous model [15], which makes a stronger assumption, namely that after a certain *global stabilization time*, all message are delivered within a known time Δ and are none of them are lost. However, the question is how to obtain this reliability of practise. As evidenced by our observations of the network deadlock, even the use of protocols like TCP does not necessarily solve the problem completely under finite memory constraints.

On the other hand, trying to resolve the problem by dropping messages when buffers become full would violate the reliable delivery property and require handling of the corner cases described in section 3.4 (and potentially more). However, for all practical purposes of our experiments, the problem was solvable by increasing the networking buffer sizes sufficiently.

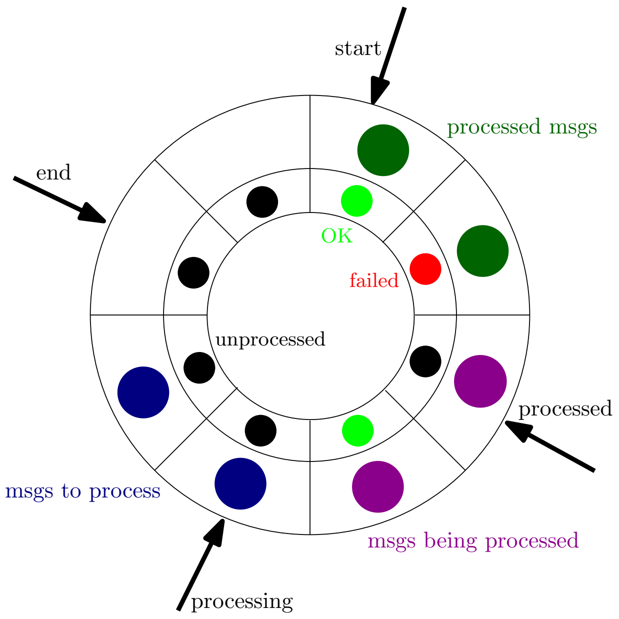


Figure 3.4: Overview of our cryptographic buffer. Bytes belonging to messages are depicted on the outside and flags on the inside. The arrows outside the ring represent the `start`, `processed`, `processing` and `end` indices.

3.3 Cryptography

We will now discuss how cryptographic operations are used in our implementation. We use SHA256 to compute MACs and RSA on SHA256 digests for public key signatures. To avoid having to implement these operations ourselves, we use the OpenSSL library [38].

3.3.1 Signature structure

This section describes the types of signature used, their structure, the parsing of this structure and which parts of the message they sign. We use public key signatures and MAC authenticators. A public key signature is a constant-size byte segment, which requires no structure parsing from our implementation. An authenticator consists of n MACs, n being the number of servers. The entries are consecutive in memory with no information about its structure explicitly represented. Messages sent to servers have all entries filled with MACs authenticating the message for each of the servers. If replies to clients are signed with authenticators, they only have the first entry filled with a MAC for the recipient.

The signature authenticates the header and the digest but not the payload, whose authenticity can be verified by checking its actual digest against the digest specified in the message. This is needed to enable sending of pre-prepare messages in view-changes without having to send their payload (which would make the view-change messages too large).

To compute the length of a signature, a message recipient first finds out the signature type from its command line arguments. Public key signatures are always of the same size. And as MACs are of constant size, the size of an authenticator is determined by the number of peers, which is also known to the node.

If public key signatures are used, no parsing of the signature structure is needed. If an authenticator is used, it composes of concatenated MACs of fixed length. Therefore, the start of a particular MAC can be obtained by adding the correct multiple of MAC size to the authenticator start.

3.3.2 Cryptographic buffer design

We use a single cryptographic buffer for signing and verification. For simplicity, we will use verification to illustrate the logic common to signing and verification and then describe how the buffer is configured to perform either one or the other operation.

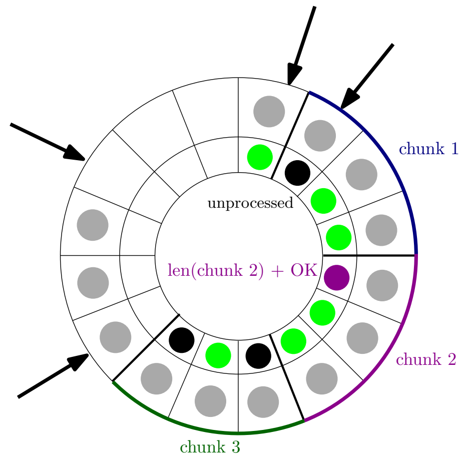
The buffer is depicted in figure 3.4 and consists, at its core, of a block of memory and four indices. The block of memory has the message bytes inserted into it and is interpreted as a circular buffer. Two of the indices indicate the start and end of the valid segment in the buffer. The remaining two indices, `processed` and `processing`, indicate the start and end of the segment being currently verified by some thread. Therefore the logical interval [`start`, `processed`) contains verified messages that should be removed from the buffer, [`processed`, `processing`) contains messages that some thread is working on and [`processing`, `end`) contains messages that can be picked up by an available thread to work on. The buffer has a single lock to protect all the indices and a condition variable used to wait for the indices to be advanced (for example, the cryptographic threads may wait for `end` to be advanced so that they have some messages to verify).

Threads verify the messages in chunks of approximately the same size (measured in bytes). More precisely, there is a threshold of b bytes. If there are at least b bytes available, a thread will pick a chunk of the smallest possible size $B \geq b$ such that the chunk contains a whole number of messages. If there are less than b bytes available, the thread will choose all the available bytes as its chunk. Each thread verifies its chunk, obtains the lock, advances the `processed` index as appropriate, picks a new chunk, releases the lock and starts verifying the new chunk.

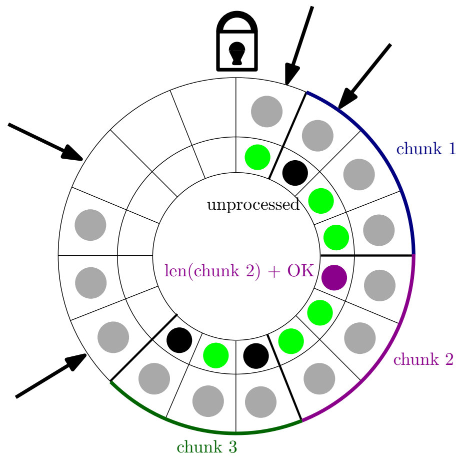
Furthermore, the buffer also contains a set of flags that indicate when each message has been verified and the result of the verification. Each flag has three states — `UNPROCESSED`, `OK` and `FAILED`. Messages can be verified out-of-order — for instance, the message at location [`processed`, `processed` + `msglen`) need not be verified first. If another message is verified before it, the `processed` index cannot be advanced immediately and when the message starting at `processed` is finally verified, the index may need to be advanced by more than the single message length. Flags help with this process, as is depicted in figure 3.5. As a thread is verifying messages, it sets the corresponding flags to `OK` or `FAILED`. When it finishes processing a chunk, it locks and tries to advance the `processed` index. This can be done exactly when the chunk it just verified starts at `processed`. If this is the case, it will advance `processed` first to the start of the next chunk and then keep advancing it until it reaches a flag set to `UNPROCESSED`. If `processed` cannot be advanced, the thread will simply proceed and choose its next chunk, leaving `processed` to be advanced by other threads based on the flag values.

However, the way of flag handling described above would lead to data race on the flags because threads don't hold the lock while setting these flags. A thread A could be changing a flag inside its chunk from `UNPROCESSED` to `OK` and another thread B could be advancing the `processed` index at the same time and read this flag. This could happen because only thread B needs to hold the lock for its operation, and thus would create a data race. To mitigate this issue, we do not let a thread write the first flag of its chunk until it has processed the entire chunk and obtained the lock again. The first message in the chunk is verified first and without holding the lock but instead of writing the result immediately to the flag, we store it in a local variable to be written to the flag later. This way, thread B can't advance `processed` into thread A's chunk before A sets its first flag. Until then, B will only read the first flag of A's chunk, so the only flag there could be a race on is this first flag itself. However, both threads only access this flag while holding the lock, and thus there is no race.

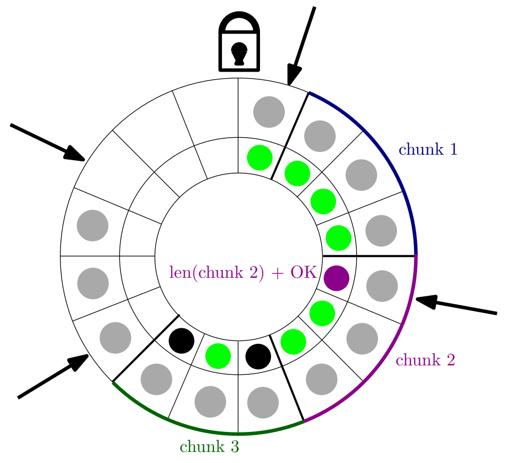
Furthermore, we found that advancing `processed` by one message at a time while holding the lock was slow. Because of our handling of the potential data race describe above, the index could be advanced not on message granularity but on granularity of entire chunks (if the first flag of a chunk is set to `OK` or `FAILED`, then all other flags in that chunk are too and the index can be advanced past the chunk). However, the thread advancing `processed` needs to know the exact length of the chunk for this be viable. To achieve this, we encode the length of each chunk into the first flag in it. The lowest 2 bits encode the state of the flag (`UNPROCESSED`, `OK` or `FAILED`) and the remaining bits encode the length of the chunk. If the flag is still set to `UNPROCESSED`, the length is not set and remains 0. The length and the flag state can then be extracted by bit shift and a bitwise AND respectively.



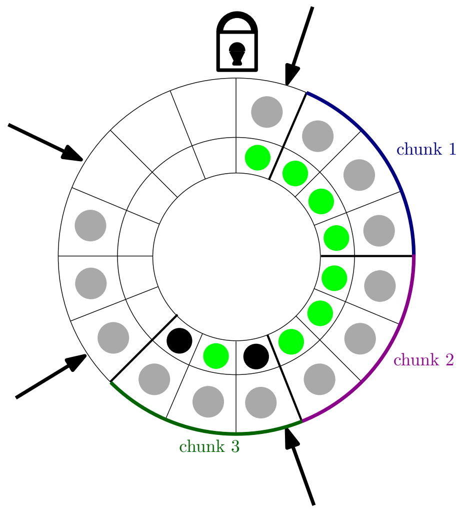
(a) Thread processing chunk 1 keeps the first flag set to `unprocessed`.



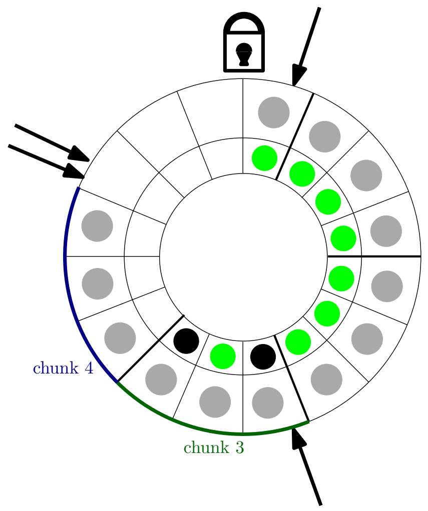
(b) The thread obtains the lock before setting the first flag in its chunk.



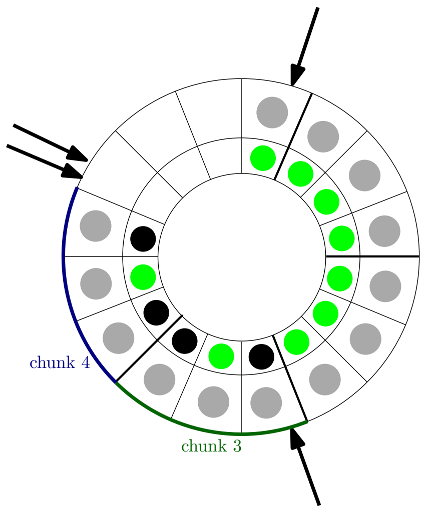
(c) Then the `processed` index is advanced to the start of chunk 2 and the corresponding flag checked. It is either set to `unprocessed`, or has the chunk length encoded in it.



(d) Since chunk 2 has already been processed, the `processed` index is advanced further.



(e) The thread then gets the next chunk advancing `processing` as it does so.



(f) Only then does it release the lock and starts processing the new chunk.

Figure 3.5: Advancing of the `processed` index in our cryptographic buffer.

Behavior customization The cryptographic buffer can perform either signatures or verification and this also affects the source of messages coming into the buffer as well as the destination of processed messages. To enable this flexibility, the buffer has to be provided with callbacks for processing (signing or verifying) a single message and getting messages to place in the buffer. In order to facilitate the use of processing callback, we unified the signature of the signing and verification functions. Messages can be placed into the buffer either by the provided callback, or by calling a push function. In the latter case, the callback should be set to a provided function that waits for new messages to arrive. Processed messages are removed from the buffer by calling a pop function.

Pushing and popping The pushing and popping functions work in a similar way to those present in the networking buffers. There are `push_start`, `push_end`, `pop_start` and `pop_end` indices present, which are accessed only by the thread calling the push or pop operation. Once a significant number of bytes is pushed into the buffer, the pushing thread obtains the buffer lock and sets `end` to the value contained in `push_end` and updates its value of `push_start` using the `start` pointer. The same operation is performed when the pushing thread sees (based on the value in `push_start`) that there is not enough space in the buffer. If there is no space in the buffer even after updating `push_start`, the push operation waits using the buffer's condition variable for the `start` index to be advanced. The converse holds for the pop operation.

Message fetching callback Besides pushing, messages can be fetched by a provided callback `get_msgs()`. Since this callback may not be thread-safe, we only call it from one thread at a time. We will now describe the synchronization needed. For the case when messages are supplied by the push operation, we provide an implementation of this callback that waits for messages to be pushed into the buffer. For this reason, the thread fetching the messages using the callback can't hold the buffer lock as this would hinder the push operation. Instead, there is a flag, `receiving`, denoting the fact that messages are already being fetched. When a thread A attempts to obtain a chunk to process, it locks. If it finds that there is no data to be processed, it sets this flag, releases the lock and calls `get_msgs()`. If a second thread also B finds there is no data to process, it finds the `receiving` flag is set and instead waits for thread A to fetch the messages. When thread A completes fetching, it obtains the lock again and clears the `receiving` flag. Then it signals a condition variable that thread B has been waiting on and processing can continue.

Flags representation In our buffer, one flag per message is needed. However, messages can have different sizes, meaning the buffer may be able to accommodate different numbers of messages depending on their lengths. The varying message size is also a challenge for designing the mapping between the message location in memory and its corresponding flag location. A simple design would be to have a flag for each byte of main buffer and for a message starting at an offset of o bytes from the buffer start, use the flag at an offset of o bytes from the flag segment start. Since messages have more than one byte, some flags would not be used. This design would however require allocating just as much memory for the flags as for the messages, which would be wasteful. Instead, we use the fact that there is a lower bound to the message start as each message has a header and either a public key signature or an authenticator. Therefore, we can compute this minimum message size s and allocate s times less memory for the flags. For a message starting at offset o, we then use the flag at offset $\lfloor o/s \rfloor$. Since the starts of each two messages differ by at least s, they will be rounded to different flag offsets and the messages will use different flags. Some flags still remain unused when larger messages are encountered but the memory required for the flags reduces by a factor of more than 100x.

3.3.3 Data race freedom

In the section, we argue the absence of data race in the cryptographic buffer. The `start`, `processed`, `processing` and `end` indices are only accessed when holding the buffer lock. The `push_start`, `push_end`, `pop_start` and `pop_end` are each accessed by only one thread. The situation is more interesting when considering the flags and the buffer bytes themselves. Since the access pattern of message bytes is almost the same as that of flags, we will mostly only argue data race freedom on flags. The same argument applies to the message bytes themselves.

We can divide the flags in logical intervals [`start`, `processed`), [`processed`, `processing`), [`processing`, `end`) and [`end`, `start`). If a thread A is accessing a certain flag when it is n an interval I_1 and a thread B is accessing the same variable when it is in a different interval I_2, then one of the indices `start`, `processed`, `processing`, `end` must have been advanced to move the flag from I_1 to I_2. Consider this movement to be done by some thread C. Thread C must have held the buffer's lock when advancing the index and the unlock operation it performed synchronizes with the lock done by thread B to read the index (and conclude the flag is in I_2). Therefore thread C advancing the index happens-before any operations done by thread B. Thus, it is sufficient to argue two things. First, that the flags are accessed only by one thread at a time while in a particular interval I. And second, that when a thread C is moving flags from interval I_1 to I_2 by advancing an index, all operations done by any thread A on I_1 happen-before the index movement.

Flags corresponding to messages in the logical interval [`start`, `processed`) are only accessed by the pop operation, which, just like the pop operation, is only called from one thread. Flags of messages in [`processing`, `end`) are not accessed while in this interval. Each flag in [`processed`, `processing`) belongs to a chunk of some thread and no two chunks may overlap (this is ensured by holding the lock while deciding on the chunk boundaries). Since each thread only accesses its own chunk, [2] data race freedom between threads performing the cryptographic operation is assured. Some of the flags in [`end`, `start`) are set to `UNPROCESSED` as new messages are placed in the buffer. This can be done in one of two ways — either the push operation is called, or one of the cryptographic threads receives the data using a provided callback. In the former case, data race freedom is ensured by the pop operation only being called from one thread. In the latter case, only one thread is allowed to perform the receiving at a time. This is ensured by raising a flag called `receiving` while holding the buffer lock. The thread that raises the flag does the receive, while other threads wait.

Next, we need to argue that when an index is advanced and thus flags moved from I_1 to I_2, all operations done on I_1 happen-before the index movement. We will again split the argument by the four logical intervals. There are no operations done on [`processed`, `end`). The message insertion done on [`end`, `start`) happens-before the advancement of `end` because both are done by the same thread. The same holds for reading the flags in [`start`, `processed`) before advancing `start`. We will now focus on the interval [`processed`, `processing`) and the advancement of `processed`. Consider a thread A working on its chunk and a thread B advancing the index. Thread A will keep the first flag of its chunk set to `UNPROCESSED` until it has finished processing the entire chunk. Afterwards, it will set the flag to a value encoding the result of the operation (`OK` or `FAILED`) as well as the chunk length. Meanwhile, thread B iterates the following: it starts looking at the first flag of a chunk. If that flag is set to `UNPROCESSED`, it stops advancing the index. Otherwise, it decodes the chunk length and sets `processed` to the start of the next chunk. Therefore it only reads the first flag of each chunk and only while holding the lock. At some point, thread B may look at the first flag of thread A's chunk. Since both threads only access that flag while holding the lock, no data race is formed by merely B reading it. If B advances `processed` to the next chunk, then it must have read the value A wrote to it after processing its chunk. More precisely, the operations of A on its chunk happen-before A writing the first flag. During the write it is holding the buffer lock and the subsequent unlock operation synchronizes with the lock performed by B before reading the value, which in turn happens-before B advancing `processed`. Thus any operations performed by A on its chunk happen-before B advances the index past the chunk, and data race freedom is therefore achieved.

3.3.4 Speculative digest checking

Replicas are to check the digest of received prepare and commit messages against the pre-prepare message. However, we have observed that a node can receive a prepare message from its peer before the pre-prepare message from the leader arrives. In this case, the correct digest is not available. To avoid having to delay the handling of these messages until the pre-prepare message arrives, we propose a method of speculative digest checking, which avoids this delay if all digests are correct.

When the first prepare message comes and the pre-prepare hasn't arrived, we store the digest and use it to verify digests of subsequent messages. If all the digests are correct, eventually the node will receive the pre-prepare message and verify this fact. If one of the received digests is wrong, the node will use the fact that it has stored the prepare messages in a log. It will zero its counter of prepare and commit messages and calculate the correct values of these counters by replaying the received messages using its log (and checking their digests against the now available digest from the corresponding pre-prepare message). As this may

[2]Except when advancing the `processed` index, in which case, the thread can only access the first flag of some chunk and these flags are protected by the buffer lock.

result in the counter of correct prepare messages dropping below $2f$, the replica refrains from sending a commit message before actually receiving the pre-prepare for the corresponding request.

3.4 State-machine module

The state-machine module is in charge of interpreting received verified messages and generating messages that should be signed and sent. It is responsible for assigning ids to incoming requests on the leaders, counting prepare, commit and other votes, maintaining message logs, etc.

Most of the data kept is stored on a per-request basis. Information about old requests can be discarded when a stable checkpoint is created, in which case we can also discard data for all older requests. Moreover, there is a maximum number of requests whose data needs to be kept at once (because nodes reject requests more than k checkpoint intervals above their last stable checkpoint, where k is a constant). This allows keeping this data in a circular buffer by their request ids, which was done by moduBFT [43]. We took a similar approach. There is a circular buffer for counting prepare messages, one for counting commit messages, etc. The message log is also implemented by circular buffers — one for each appropriate message type. Circular buffers are also used for storing checkpoint votes and their counters, although these buffers are significantly smaller.

When a node is broadcasting a prepare or commit message, it sends a copy to itself too. To avoid having to verify this message and moving it from one buffer to the next, we extracted the heart of message handling into a dedicated function, `handle_msg()`. This function gets a single message, assumes it has been verified and acts accordingly, possibly generating new messages. When it generates a prepare or commit message to be broadcast, it calls itself with this new message as an argument. Thus the message does not have to pass through the regular sign-network-verify path.

The most demanding part of implementing the state-machine module is dealing with corner cases. There are many corner cases in the PBFT protocol, particularly when attacks are allowed, which substantially increases required implementation complexity. These corner cases usually manifest in the state-machine module. In the rest of this section, we will describe how we dealt with some of them and the architectural decisions driven by their existence. The focus of this work is efficiency of normal operations (i.e. no view changes are necessary). For this reason, we were forced to abandon handling of some of these corner cases to keep the development effort manageable. Although this poses some limitations to our resulting program making it a prototype rather than full implementation, it allowed us to focus on optimizing the normal operations.

3.4.1 Reliable message delivery

The original PBTF paper [11] states that the protocol works with UDP even when messages can be dropped or delivered out of order by the network or delayed by the adversary. However, allowing messages to be dropped increases the problem complexity, particularly for view changes because there is no attempt to resend an old view change message for change to view $v + 1$. Thus, problems can arise even if we assume that if a message is constantly being resent, it will eventually be delivered.

Consider a scenario where the client doesn't receive a sufficient number of matching answers in time due to the answers being lost by the network and it resubmits the request (by broadcasting it to all replicas) but f of the working (i.e. not faulty) replicas, including the current leader, don't receive it. The adversary can be controlling f nodes but making them "cooperate" at this stage, i.e. follow the protocol as if they were not faulty. The $f + 1$ working replicas forward the request to the leader but they also get lost; and they start their timer. When the timer expires, the $f + 1$ working replicas start a view change and the f faulty nodes join them, resulting in $2f + 1$ view-change messages, which is sufficient for the view change to go through. The leader of the new view $v + 1$ broadcasts a new-view message but the f working replicas who missed the request resubmit also miss this message and don't change their view. This makes them decline message sent in the new view and effectively become unresponsive. If the adversary makes its f faulty nodes become unresponsive as well at this stage, the remaining $f + 1$ replicas are unable to commit any messages. That in turn makes the client resubmit its requests and view changes to start.

However $f + 1$ of the working replicas will try to change view to $v + 2$ while the remaining f to $v + 1$ (and the faulty nodes can simply be unresponsive forever). Therefore neither group is able to receive $2f + 1$

votes for the view change it is trying to push through. Consequently, no node starts its view-change timer and never try to change to a view higher than $v + 1$ resp. $v + 2$ and get stuck in the view-change process. This makes the system unresponsive. If the replicas stop accepting all except checkpoint, view-change an new-view messages, any resubmits from the client will be ignored.

We expect that this scenario could be handled by a carefully thought-through implementation. Perhaps this situation could be resolved by monitoring resubmits from clients even when trying to change view, or by not requiring $2f + 1$ view-change message *to the same view* to start the timer and instead reacting appropriately to $2f+1$ view-change messages which are however to different views. Nevertheless, we wanted to avoid having to reason about such corner-cases to be able to devote more effort to performance optimizations.

Another example of our concerns would be what will happen if a replica misses too many checkpoint messages and therefore is unable to create a stable checkpoint, which will eventually make it reject messages with request id set too high above its last known checkpoint.

For these reasons, we decided to rely on a reliable transport protocol, namely the `AF_STREAM` Linux sockets. They also have the advantage of assuring ordered delivery at the same time, which also simplifies implementation as we will describe later. The same decision was made by moduBFT [43], which also uses TCP.

3.4.2 Request timestamps

Client's request are to include so called *timestamps*, whose original purpose is to guarantee exactly-once semantics. This is not a concern for us due to our choice of the delivery layer, this not a concern for us. However, we still use timestamps when a request is resubmitted to identify the request id it was previously assigned. To do this, the client keeps the timestamp of a resubmit the same as the timestamp of the original message and the servers maintain a timestamp-to-request-id mapping.

The PBFT paper only requires the timestamps to increase monotonically for each client and proposes the use of the client's local clock. However this complicates maintaining the mapping to request ids as the timestamps may not be consecutive, especially if messages can be delivered out of order. The timestamps could be kept in a binary tree but the insertion required would be costly and since it would have to be performed with every received request even during normal operations, it would likely slow the implementation down. Alternatively, they could be stored in an array in the order in which the requests came. Since this order is identical to the increasing-request-id order, old stored timestamps could be easily discarded when a checkpoint is created (because they would form a continuous segment in the array). However, to identify whether a request has been previously received would require a sequential search through this array if requests can be delivered out of order, or a binary search if ordered delivery is guaranteed.

To improve upon this, instead of the client's local clock value, we use the request sequential number (i.e. each client's first request will have timestamp equal to 0, second to 1, etc.). This way, the entries can be kept in an array for each client — the id of the request with timestamp t being in the array at index t. Discarding old data now requires ordered delivery but there are other reasons for relying on it, which will be described later. Also, the `SOCK_STREAM` sockets we use for reliable delivery, ensure ordered delivery too. On the positive side, insertion in the array can now be done in constant time and so can the lookup.

3.4.3 Ordered message delivery

Similarly to unreliable message transport, the possibility of out-of-order delivery poses challenges to the protocol implementation. A message from a client to the leader may be resubmitted even if executed properly by the replicas because the replies to the client may become delayed. To avoid re-executing requests, servers are to cache their answers and resend them if needed. However for that to happen, they need to identify which answer the resubmitted request belongs to. And as nodes have only finite amount of memory, the data allowing this identification needs to be discarded at some point. The paper has the data regarding a request r discarded when a stable checkpoint is created with a higher request id than r. However, the creation of said checkpoint doesn't mean the client already received the answer to that request and hasn't resubmitted it. Consequently, it is conceivable that a resubmit arrives for a request whose data has already been discarded. Since the client may not know the id of the resubmitted request (the id is assigned by the leader, and thus the client cannot know it if it received no answers), it is hard to tell this situation apart from one where the resubmit arrived before the original request.

In such a situation, we would like to assume that one of the following happened:

- the request was already executed and the client will receive its replies eventually (as we have reliable delivery and the answers must have been sent to create the checkpoint that discarded the requests data)
- the original request has not been received yet but it will eventually (due to reliable delivery)

and consequently, the request doesn't have to be executed now. However, this assumption doesn't hold because when the leader experiences too high load and the number of pending requests becomes too high for its memory constraints, it needs to start dropping incoming requests. The reference implementation we use, moduBFT [43] uses circular buffers of fixed size to keep request data and silently drops requests that do not fit into the buffer. In such a case, the requests need to be resubmitted by the client, in which case when the resubmit arrives, there will be no data for it even though the original request arrived previously and wasn't executed.

On the other hand, if we execute all resubmits where there is no data for the original request, we risk executing a request multiple times if it is resubmitted and the resubmit arrives before the original request. This repeated execution may be incorrect for requests that modify the service's state.

The issue is with representing the set of requests that have been accepted from each client — if we knew this set, we could identify whether the resubmit belongs to it (and we should ignore it) or doesn't (and we should treat it as a new request). We can't represent this set indefinitely since we have only finite memory. We could attempt to represent it by storing for each client the highest timestamp accepted from that client. Unfortunately, that approach also fails because if messages can be delivered out of order, the leader may get a request with timestamp T first and only afterwards receive a request with timestamp $t < T$. In such a case, it would incorrectly assume the request with timestamp t belongs to the set.

Ordered delivery prevents many of these pathological situations, most notably a resubmit of a request cannot arrive before the original request. The only problem occurs when requests are dropped due to high load. In this case there is no data kept for the request and when the resubmit comes, it will appear to the leader as if the original request never came at all. The advantage is that the adversary can no longer create this situation without getting control over the clients. Furthermore, we expect this could be fixed if needed with a strategy we will describe next. We decided not to implement this fix due to time considerations and because we decided to prevent request dropping altogether in our experiments.

The timestamps sent by the clients are consecutive and we have reliable ordered delivery. Therefore, if the leader last received timestamp t from a certain client, it should receive one with timestamp $t + 1$ next time. If the next timestamp is higher, the request with $t + 1$ must have been dropped. Thus, we could leverage the approach described previously of keeping the highest received timestamp for each client to represent the set of received timestamps. Thanks to the ordered delivery, this approach is be correct as long as the leader doesn't accept the request with timestamp $t+2$ before receiving a resubmit of the one with $t+1$. This could slow down progress after request dropping as no other requests from the affected client could be executed before resubmitting but during normal operations (with no requests dropped), the only overhead would stem from maintaining the integers representing the highest timestamp received from each client.

3.4.4 Number of view-change counters

The PBFT paper states that when the leader of a new view receives $2f$ view-change messages for this view, it should send a new-view message. Other replicas are supposed to wait for $2f + 1$ view-change messages and then start their view-change timers. However, the paper doesn't state how replicas should keep track of the number of received messages and there may be some votes to move to view $v + 1$, $v + 2$, etc. How many counters at the most could be needed?

The adversary is able to force view changes by delaying network messages. Namely, he can delay all the messages from the current leader making it appear unresponsive to the other replicas, who then force a view change before the delayed messages are delivered. If the current leader is l, the adversary can choose to attack peer $p = l + 2$ to make it use many view change counters. He will hold up all messages to p until otherwise noted and force a view change to $p - 1$. Since p doesn't receive the new-view message (it is held up), it will be unsuccessful in trying to change view to $p - 1$. The adversary can then force more view changes. First an unsuccessful change to p and when p doesn't send the new-view message in time (because it does not receive the view-change votes), the view will be changed to $p + 1$. The adversary can then force a change of leader to $p + 2$, $p + 3$, $\ldots$, $p - 2$, $p - 1$, etc. Each time, he first makes the leader appear unresponsive, lets the view change happen and delivers all delayed messages, except those for peer p. This can keep happening as long as the adversary can delay messages for p. Then, he can deliver all held-up messages from $p - 2$ to p at once. Thanks to ordered delivery, this sequence is guaranteed to contain the new-view message from $p - 2$ after the view-change votes for views $p - 1$, p, $p + 1$, $\ldots$, $p - 3$ but this requires $n - 1$ counters. If ordered delivery were not assured, he could also permute the messages so that any new-views arrive last, requiring potentially unbounded number of counters. This is another reason for choosing ordered delivery.

In practise, we do not expect such corner cases to happen but to allow for some defense against a replica missing a number of view-changes, we implemented support for having an arbitrary constant number of view-change counters.

3.4.5 Replicas falling behind

Since there is only limited memory available on the nodes, there can be only a finite number of requests being processed at once. PBFT [11] has replicas ignore any messages with request id to high above the last stable checkpoint. All information belonging to requests before the checkpoint is freed to be reused. Like moduBFT [43], we implement this by storing this information in a circular buffer based on the request id and making the buffer large enough to accommodate all messages between the last checkpoint and the high watermark, beyond which messages are rejected. If the leader receives a requests, it tries to assign it an id. If this id is too high (above the high watermark), it drops the request.

Also, like moduBFT, we measured our performance using a constant *asynchronous factor* — the maximum number of requests a single client is willing to have pending at any point in time. We expected that keeping this asynchronous factor low enough would allow for all pending requests to fit into the circular buffer, thus preventing dropping of requests. However, we still observed requests being dropped. This is our explanation of how this could occur.

The client submits a request to the leader and waits for $2f + 1$ matching replies before accepting them as correct and issuing a new request to keep a certain number of pending requests. However, this set of replies doesn't have to include the leader. Therefore if the leader is not among the $2f + 1$ fastest nodes to answer, it will observe more pending requests than the clients because it is unaware that some requests have already been committed by a sufficient number of followers. Theoretically, there is no bound to how many finished requests the leader may be unaware of, except the maximum number of requests it is willing to have pending. Therefore, if the leader falls behind by a sufficient amount, it can start dropping requests no matter the asynchronous factor.

This problem can also affect followers — consider a follower whose message processing is unbalanced, giving higher priority to messages from the leader than other nodes. Let us denote the id of the last stable checkpoint known to this follower as c the checkpoint interval length as i and the circular buffer size as b. Then the client may receive pre-prepare, commit and checkpoint messages from the leader for requests with ids higher than $c + b$ without being able to advance its last stable checkpoint farther than c because it starts processing these messages from the leader before processing checkpoint messages from $c + i$ from other nodes. This can make an hones follower become unresponsive.

Creating unresponsive nodes has a potential for an attack where the adversary's faulty nodes act just like honest nodes would until a time when some honest nodes become unresponsive. At that point faulty nodes can become unresponsive too, creating more than f unresponsive nodes.

We expect that in this case, the situation could be resolved by a view change but to avoid the need to reason about this corner case, we decided to prevent unresponsive node creation altogether. We also wanted to avoid the need for the leader to drop messages, forcing the client to resubmit. Besides possible performance impacts, this would make the implementation hard to debug. To this end, we designed the following client-throttling scheme.

Every client has a constant asynchronous factor and instead of waiting for $2f + 1$ replies, it waits for replies from all replicas. To keep the fault tolerance property of the protocol, the client only waits until the request's resubmit timer expires. At that point, if at least $2f + 1$ matching replies have been received, it accepts them. Otherwise it resubmits the request per the PBFT protocol. In an environment where nodes are often faulty but performance is desired, this feature can either be easily turned off (accepting the potential for requests being dropped and resubmitted) or the time before accepting $2f + 1$ replies shortened. Furthermore, in our experiments, each client knows the value of the circular buffer b and the number of clients c and is only willing to have b/c requests pending.

3.5 Module communication

This section describes how the three modules are integrated and offers more details of the flow of messages through the implementation than was presented in figure 3.1. It also deals with our strategy of buffer flushing, which makes sure all requests are answered, even if only a small number of them arrives.

Messages are first received by I/O threads and placed in the networking buffers. They are be moved from these buffers to the verify buffer by calling a dedicated function, `net_recv_msgs()`, which iterates through the receive buffers until it finds available messages (as was described in section 3.2.2). This function should not be called by multiple threads concurrently. The thread calling the function becomes the popping thread as it was called in section 3.2.2. In our implementation, `net_recv_msgs()` is specified as the message-fetching callback of the verification buffer described in section 3.3.2, and the role of the popping thread is thus performed by one of the cryptographic threads.

After their cryptographic processing, messages need to be given to the state-machine module. This is done by performing the pop operation described in section 3.3.2. The operation is called by the state-machine module. To aid future offloading and decrease the coupling between modules, the state-machine module has a single function for fetching messages, `get_msg()`, which can be edited to obtain them from the cryptographic module, the networking module directly or any other source if needed. This function returns messages one-by-one.

A message created by the state-machine module is passed to the counterpart of `get_msg()`, which normally pushes it to the sign buffer but can also be configured to skip the cryptographic module or changed to perform any other operation. After being signed, messages are popped from the networking module one-by-one by a dedicated thread, `crypttonet_thread` and given to the networking module, which moves them to the appropriate send buffers.

Flushing Because of the implementation of the push operation in the cryptographic and networking buffers, result of a single push may not be immediately visible to other threads. This is to avoid too frequent locking. However, it could lead to request being unanswered until a sufficient number of them piles up. For example if a single client is present, it would almost never receive a reply to its last request. To prevent this, we implemented a flush operation on the buffers, which is called when no requests are received for a significant length of time. We will now describe the individual flush operations as well as when and how they are performed.

At its core, the networking flush simply obtains the lock protecting the buffer, updates `end` using `local_end` and releases the lock. The flush in the sign buffer is similar, updating `end` using `push_end` while holding the lock. However, to flush all buffers properly, the network flush needs to be called after cryptographic flush was performed, the messages actually signed and placed in the send buffers. The need to perform a flush also needs to be communicated to all modules. We will now outline this communication.

When the pop operation from the verify buffer has no bytes to return, it waits but not indefinitely. If it times out, it returns without extracting any messages. When the state-machine module receives no message, it calls a flush on the sign buffer. This updates the `end` variable as mentioned before but also sets a special variable, `flushing` to the current value of the `end` index. When the messages are popped to be placed in the networking buffer, the pop operation uses the value of `flushing` to detect that and when the flush was called. If the message it is about to return is the last message that was pushed into the buffer before the flush, this is communicated to the `crypttonet_thread`. When this thread has moved this special message, it calls the networking flush.

Chapter 4

Experimental results

This chapter describes our measurements. Be begin by assessing the performance of and profiling our main reference implementation, moduBFT [43]. We also benchmarked hotstuff as a further potential comparison and present the results. However because of differences between our implementation and hotstuff, this cannot serve as a direct comparison. Lastly, we present a performance evaluation of our implementation.

All experiments were run on the ETH Heterogeneous Accelerated Compute Cluster (HACC) cluster [16] on nodes having 16-core AMD EPYC 7302 CPUs with maximum frequency 3GHz, L1 cache size 512KB, L2 8MB and L3 128MB. The nodes had a 100 Gb/s network interconnect using mellanox NICs [36]. We used Go 1.20.1 for moduBFT and GCC 9.4.0 for compiling hotstuff and OBFT. We had openSSL 1.1.1 available [38].

4.1 ModuBFT

ModuBFT provides scripts that can be used to measure the throughput and latency distribution for different message sizes, client and peer counts. It allowed us to vary client counts until the servers saturated. The script could be readily used provided clients were run on different IP addresses. However, saturation was reached for high client numbers, while HACC offered a relatively low number of nodes. This had two consequences.

First, this limited the server counts we were able to benchmark. Since we wanted to measure performance of the servers, we could not collocate multiple servers on one physical machine as this would make our results for different peer counts incomparable. For the same reason, we could not collocate servers with clients, thus requiring us to reserve one physical machine per server. Given the number of machines routinely available on the cluster, we decided to measure performance for up to 10 peers.

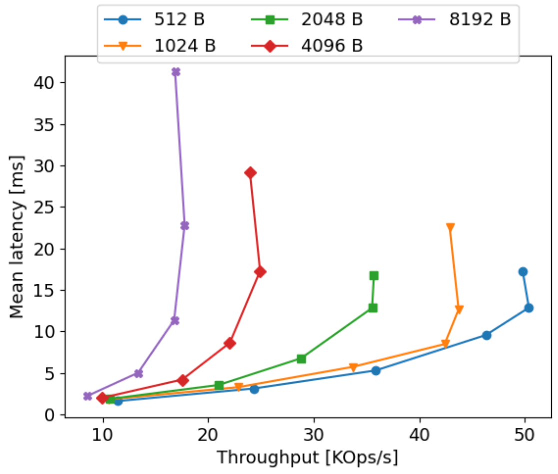
Secondly, multiple clients needed to be run on a single node because saturation was reached for too high node counts. To facilitate this, the following changes had to be made.

4.1.1 Modifications

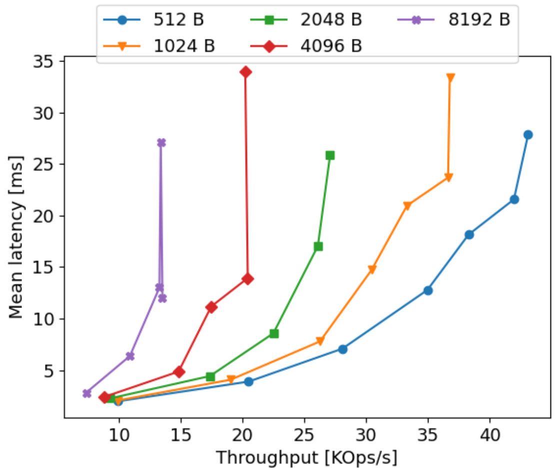
First of all, when a server receives a connection, it needs to identify which client initiated it. The original implementation used the client IP address read from the experiment configuration and assumed the connection to be coming from the first client with that IP address. The connection objects were stored in an array based on the client ID. With multiple clients sharing the same IP address, this approach could lead to entries in the array being overwritten when another client with the same IP initiated the connection. To mitigate this problem, we changed the client code to send the client ID as the first four bytes on the initiated connection and the server code to read it.

Secondly, the measurement results (latency distribution and throughput) were written to a file whose name was based on the IP address of the client measuring these values. This created a race condition allowing rewriting of results or mixing the text from multiple clients resulting in incorrect and variable results. Thus, the output file names needed to be changed in the experiment scripts.

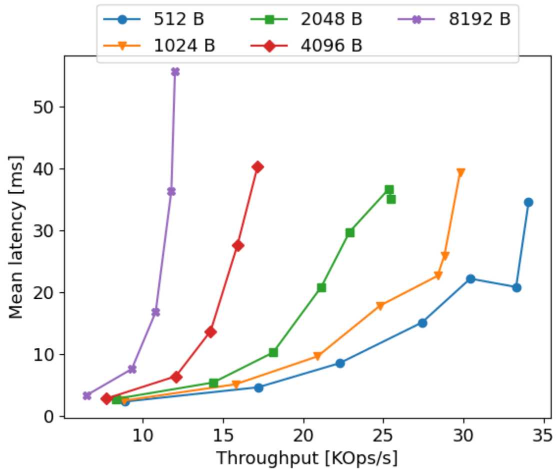
Lastly, we noticed that the original scripts as well as the performance figures in the paper report values for $2f + 1$ peers. However, PBFT, which moduBFT implements, works with $3f + 1$ peers where f is a natural



(a) 4 peers



(b) 7 peers



(c) 10 peers

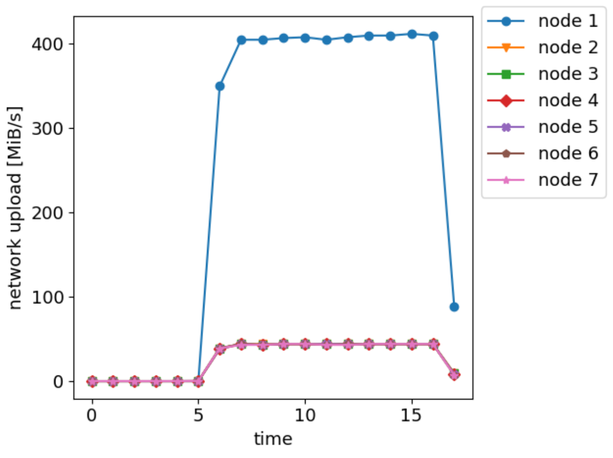
Figure 4.1: Throughput-latency graphs of moduBFT obtained by varying number of clients. Different lines represent different request sizes.

number. We ran our experiments with $3f+1$ peers (3, 7 and 10), which required minor changes in the peer code. Some of the buffers had their length calculated based on the number of peers n — an integer variable f was calculated as $f = \left\lfloor \frac{n-1}{2} \right\rfloor$ and the buffers' length was $2f+1$. Consequently, given an even number of peers, the buffers' length was $n-1$ instead of n as it should have been, which resulted in out of range errors. The buffers in questions are `Node.checkpointMsgs` and `Node.preparedLog[i]` for `i` from 1 to `CIRCULAR_BUFFER_SIZE`.

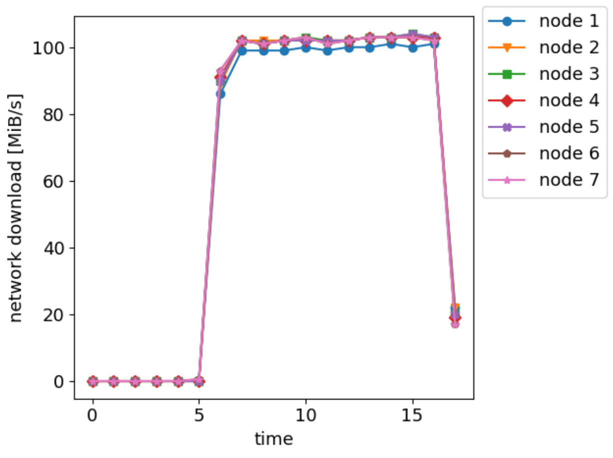
4.1.2 Results

We used the throughput and latency as our primary performance metric. The results can be found in figure 4.1.

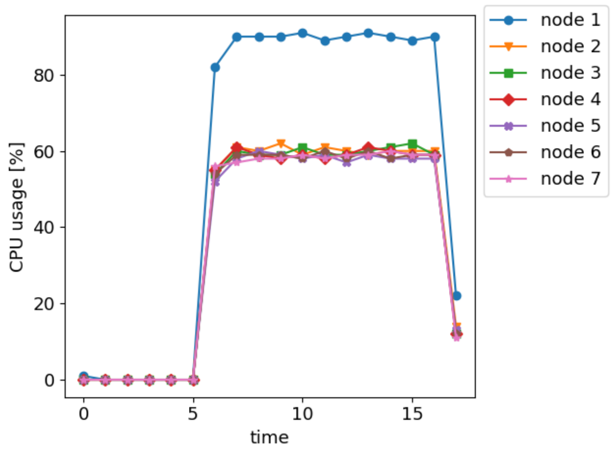
We also measured the CPU usage and network utilization to evaluate whether CPU or network operations form the bottleneck. The data was collected using the `dstat` command. As can be seen in figure 4.2, the CPU usage exceeds 80%, while network download and upload stay way below the theoretical bandwidth of 100 Gb/s.



(a) Network upload



(b) Network download



(c) CPU usage (usr+sys)

Figure 4.2: Resource usage at the point of saturation for 7 peers and message size 2048. Results for other settings are similar.

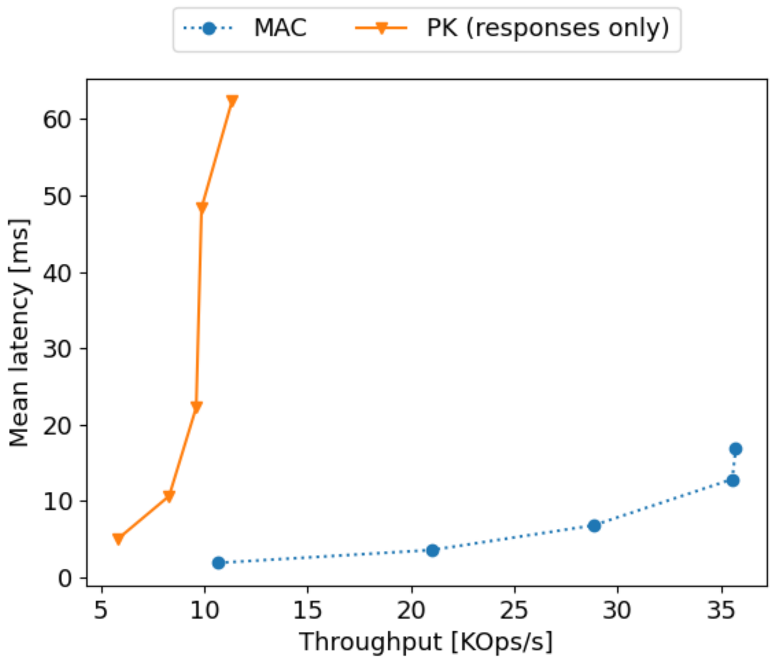


Figure 4.3: Comparison of moduBFT using private key signatures and MACs for responses to clients, other messages always use MACs.

4.1.3 Additional measurements

Since moduBFT is our primary reference implementation, we conducted additional measurements to better understand its inner workings and the bottlenecks present. Some of these measurements were attempting to make sure we are using the full potential of moduBFT and, while others influenced our design. First, we compared performance with public key signatures of peer responses to client versus using MACs for all messages. This comparison which can be found in figure 4.3. The version with MACs can achieve over 3x larger throughput compared to the more general private key signature. Therefore we knew we public key signatures to form a bottleneck and result in large performance differences.

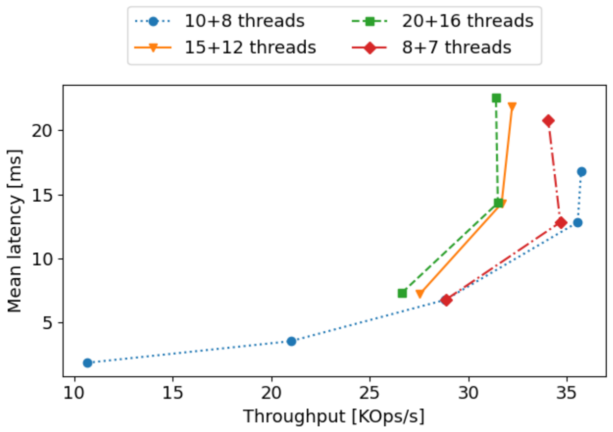
Next, we varied the number of threads[1] used by the peers for signature verification and signature creation. The default values are 10 threads for verification and 8 threads for signature creation. We tried one setting using fewer threads and two using more. Since the number of threads used for signature creation and verification was fixed in the original moduBFT code, we modified the code to read these thread counts form environmental variables and we set these variables in the benchmarking scripts. The results, presented in figure 4.4, show little difference between the different settings. There is some noticeable difference only with 4 and 7 peers and in those cases the default thread counts work well. Considering the observed CPU usage of over 80%, this suggests that the default number of threads is sufficient to efficiently utilize all available CPUs in our experiment setup.

4.1.4 Profiling

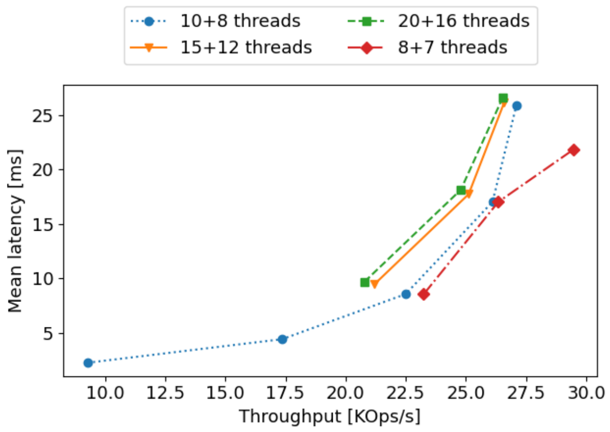
We also ran the peer program under the built-in Go profiler with different numbers of peers (4, 7 and 10) and clients (8, 16 and 24). The visual output produced by the profiler can be seen in figure 4.5. In each entry, two percentage values are shown. The first one specifies the percentage of time spent directly in the function, while the later also includes time spent in functions it called.

Notably over 45% of the time is spent in `(*Node) VerifySig` and its callees, which could be interpreted that the implementation of signature verification could be a bottleneck. However, according to the profiler,

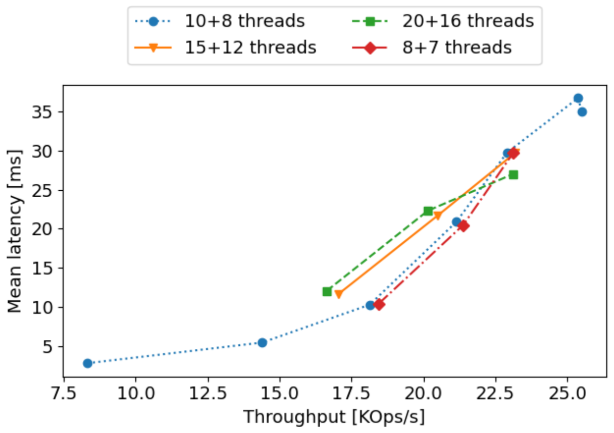
[1]more precisely, the number of goroutines



(a) 4 peers



(b) 7 peers



(c) 10 peers

Figure 4.4: Comparison of different thread counts used for signature creation and verification.

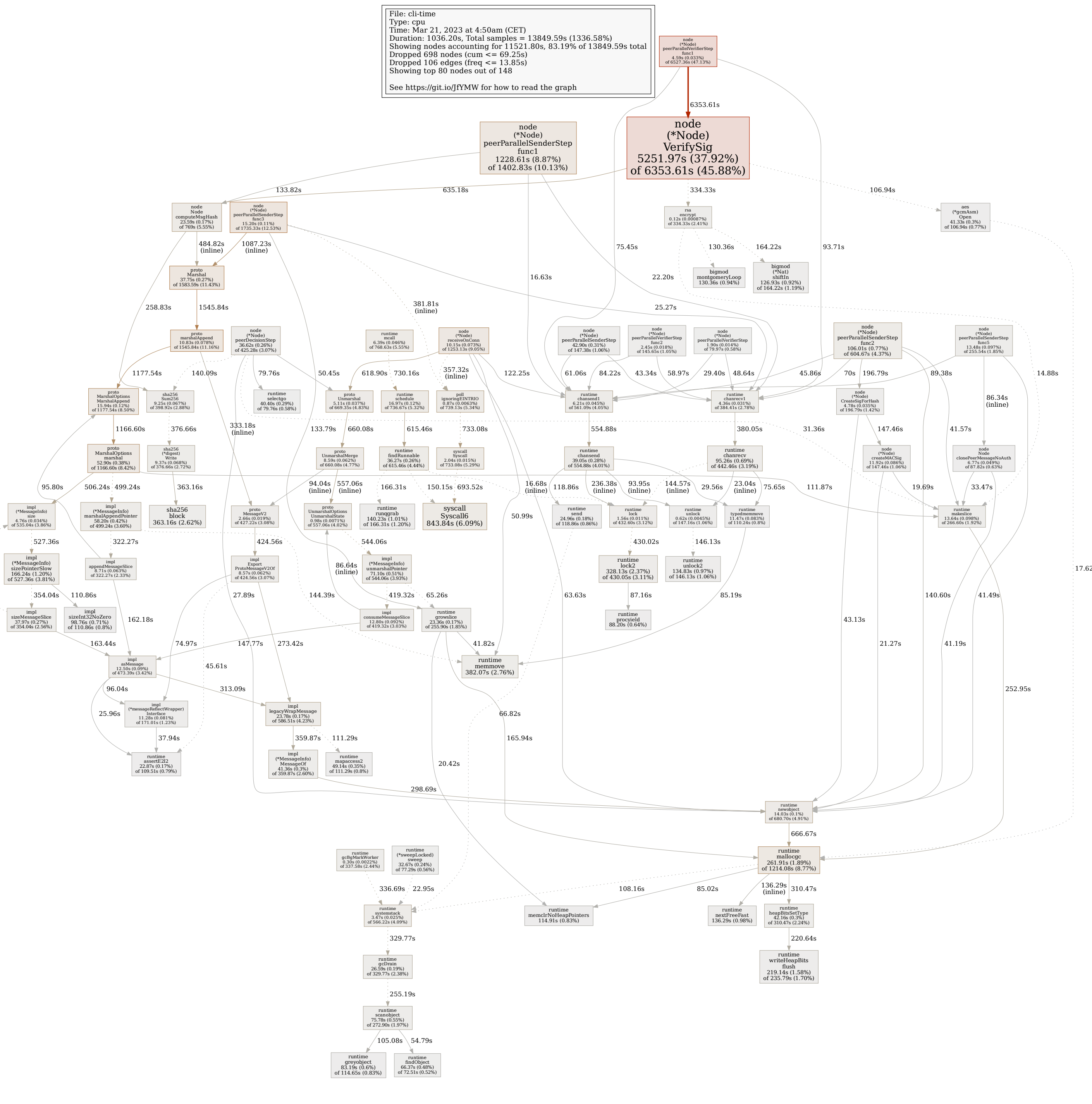
most of the time is spent in the `VerifySig` function itself, while the actual signature is calculated in the functions it calls.

The `(*Node) peerParallelSenderStep` function and nameless functions declared in it, namely `func1`, `func2`, `func3`, `func5`, also take significant amount of time. These form the sending part of the implementation described in section 2.3. Namely `func1` is responsible for computing message hashes, `func2` signs the outbound messages' hashes, `func4` takes messages from the `decOut` channel and performs round robin distribution among the hashing goroutines and finally, `func3` marshals and sends the messages. In our implementation, we abandoned this complicated flow of messages in favor of a simpler one, where a message is signed once and then copied to all appropriate output buffers.

Moreover, we can see `mallocgc` taking approximately 6% of CPU time, some time being spent in synchronization operations (`lock2`, `unlock2` together taking 4%) and 6% of the time is spent in system calls. We ran moduBFT under the strace tool to find which system calls were responsible. We found that at the point of saturation, futex was the most expensive call taking approximately 30–80% of the time (usually between 40% and 50%) and it was sometimes followed by epoll_wait. We can thus conclude that a large part of the cost of system calls is also caused by synchronization.

In our implementation, we virtually eliminated dynamic memory allocation on the critical path[2] and tried limiting the number of system calls.

[2]We only allocate more memory when encountering a message larger than any previously received message. If view-changes are not needed, this only happens once, with the first request or pre-prepare message received.


```
File: cli-time
Type: cpu
Time: Mar 21, 2023 at 4:50am (CET)
Duration: 1036.20s, Total samples = 13849.59s (1336.58%)
Showing nodes accounting for 11521.80s, 83.19% of 13849.59s total
Dropped 698 nodes (cum <= 69.25s)
Dropped 106 edges (freq <= 13.85s)
Showing top 80 nodes out of 148

See https://git.io/JfYMW for how to read the graph
```

node (*Node) peerParallelVerifierStep func1 4.59s (0.033%) of 6527.36s (47.13%)
6353.61s
node (*Node) VerifySig 5251.97s (37.92%) of 6353.61s (45.88%)
node (*Node) peerParallelSenderStep func1 1228.61s (8.87%) of 1402.83s (10.13%)
334.33s
106.94s
133.82s
635.18s
rsa encrypt
aes (*gcmAsm) Open
bigmod montgomeryLoop 130.36s (0.94%)
bigmod (*Nat) shiftIn 126.93s (0.92%) of 164.22s (1.19%)
node Node computeMsgHash
node (*Node) peerParallelSenderStep func3
proto Marshal
proto marshalAppend
proto MarshalOptions MarshalAppend
proto MarshalOptions marshal
sha256 block 363.16s (2.62%)
syscall Syscall6 843.84s (6.09%)
runtime chansend1
runtime chanrecv1
runtime chansend
runtime chanrecv
runtime lock2 328.13s (2.37%) of 430.05s (3.11%)
runtime memmove 382.07s (2.76%)
runtime newobject
runtime mallocgc 261.91s (1.89%) of 1214.08s (8.77%)
runtime memclrNoHeapPointers 114.91s (0.83%)
runtime nextFreeFast 136.29s (0.98%)
runtime heapBitsSetType
runtime writeHeapBits flush 219.14s (1.58%) of 235.79s (1.70%)
runtime systemstack
runtime gcDrain
runtime scanobject
runtime greyobject
runtime findObject

Figure 4.5: Profiler output of moduBFT peers

4.2 Hotstuff

As an additional comparison, we measured the performance of hotstuff [50]. The authors provide ansible playbooks for running a single experiment, whose parameters can be controlled by changing ansible configuration files. The nodes' IP addresses are specified in a separate file and a script is provided for transforming this list of IP addresses into a valid hotstuff configuration file used by the hotstuff executable. We used this framework and added shell scripts for running multiple experiments with varying parameters. We also had to modify the ansible playbooks as the original versions assumed to have root permissions and attempted to perform some installations. Those parts had to be removed because we did not have this level of privileges on the cluster.

We kept the default parameter values except for the following. We varied the number of peers, clients and request size. We changed the maximum number of requests each client submits to infinity to unify experimental methodology with our moduBFT experiments.

Moreover, we observed that the system tended to get stuck for small asynchronous factors (the maximum number of requests each client is willing to have pending simultaneously). We suspect this was probably caused by the use of batching — the servers process k requests at once to avoid redoing some of the operations for every request and reduce the total work required.

We experimented with different asynchronous factors. With very low factors we saw absolutely no throughput. To get reasonable throughput with 4 peers and batch size of 400, we found that we need asynchronous factor of approximately 1600 and as we increased the number of peers to 7 this number increased again to slightly above $400 * \text{num\_peers}$. Therefore, we concluded we likely need strictly more than 4000 requests in flight for 10 nodes and settled on 4200. And to make results with different peer counts comparable among themselves, we used this asynchronous factor for all our hotstuff experiments.

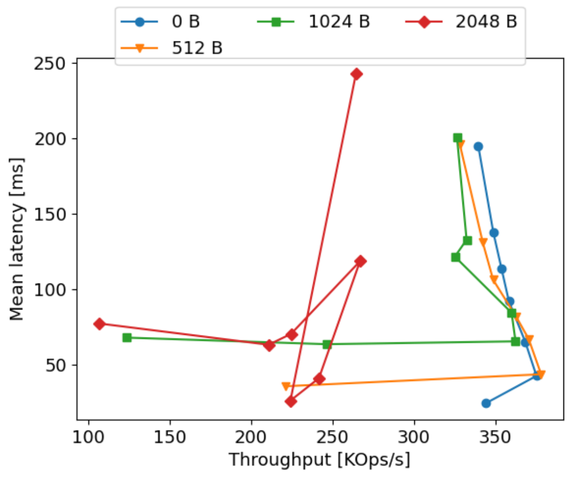
Hotstuff outputs for each individual request the time it took to process it along with a timestamp. To generate a throughput value and latency distribution similar to those used with moduBFT, we wrote a simple awk script that parses and aggregates this raw data. The hotstuff paper presents results for payload sizes 0, 128 and 1024 bytes, which is different to the moduBFT payload sizes (512, 1024, 2048, 4096, 8192). Since we wanted to compare both with our implementation, we decided to use similar sizes for hotstuff as for moduBFT and chose 512, 1024 and 2048 bytes.

Like previously with moduBFT, we needed to collocate multiple clients on each physical machine to reach saturation. Since we observed rather high CPU loads on the client machines, we used as many physical machines as practical in the cluster, which lead to approximately 4 clients per physical machine.

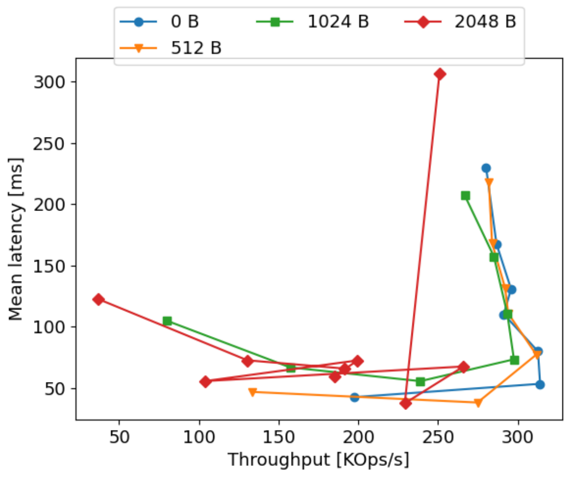
The throughput-latency graphs can be found in figure 4.6. They show more noise than our moduBFT measurements but we can see a peek throughput of approximately 350, 300 and 250 KOps/s for 4, 7 and 10 peers respectively and a base latency (before saturation) of approximately 50 to 200 ms depending largely on the number of peers.

The observed CPU usage and network utilization are presented in figures 4.2 and 4.2. They show the CPU utilization to be somewhat lower than we would expect. The precise cause of this is unknown to us. We have seen a similar combination of saturation and low CPU usage with our implementation and in section 4.3.4 we make argument for our hypothesis that this is a bottleneck in networking in our case. A similar hypothesis could explain the low CPU usage observed but would not be completely in line with the fact that when request size is increased, the measured download increases many times. Another possible reason for the low CPU utilization would be that there is insufficient number of threads to utilize the entire machine. However, when raising the number of threads, we saw no increase in performance.

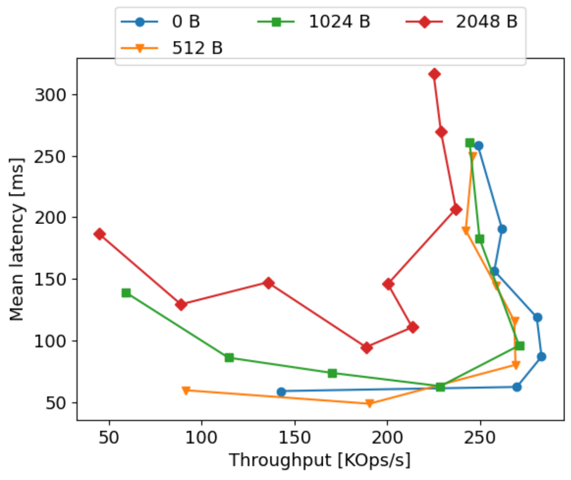
Secondly, if large requests are used, the download bandwidth observed on the peers is higher than the upload. This can be explained by the fact that each client sends its request to all replicas (which also removes the need for the peers to broadcast the request body), raising the network download on the peers.



(a) 4 peers

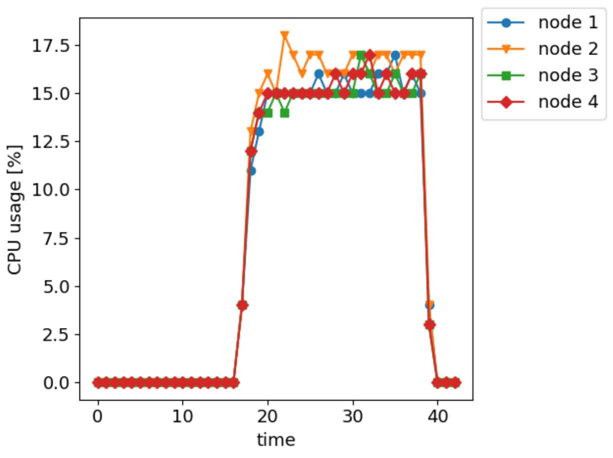


(b) 7 peers

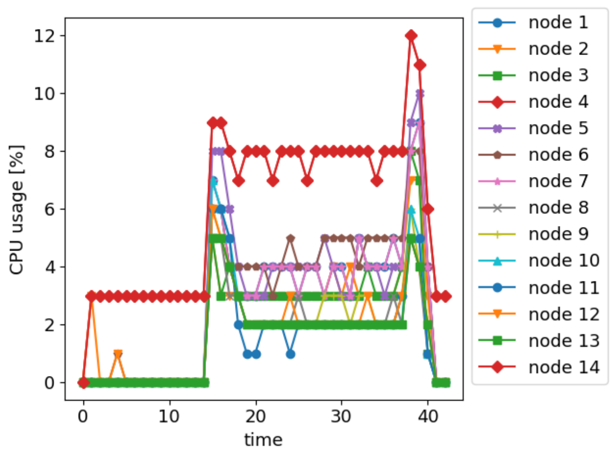


(c) 10 peers

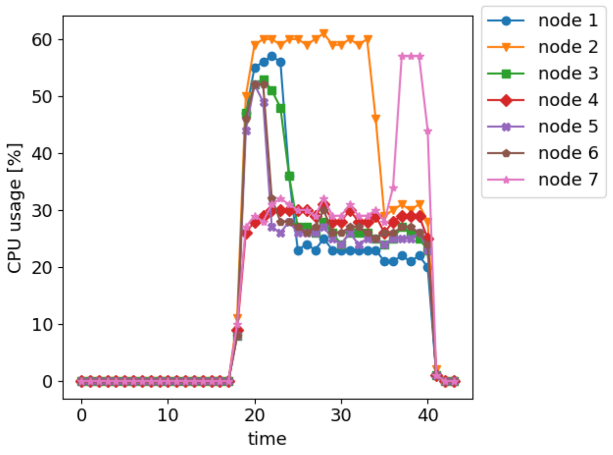
Figure 4.6: Throughput-latency graphs of hotstuff obtained by varying number of clients



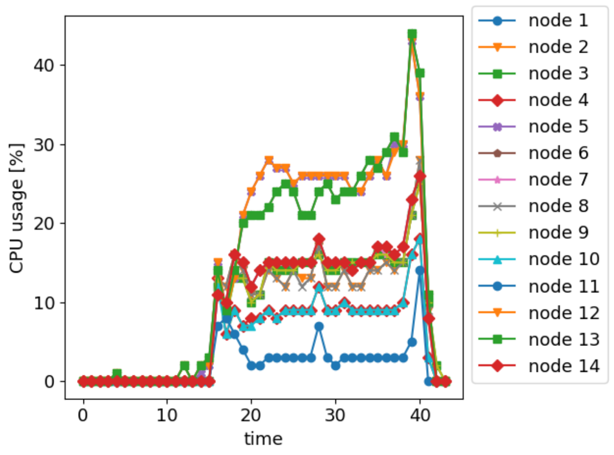
(a) servers with empty requests, 4 peers



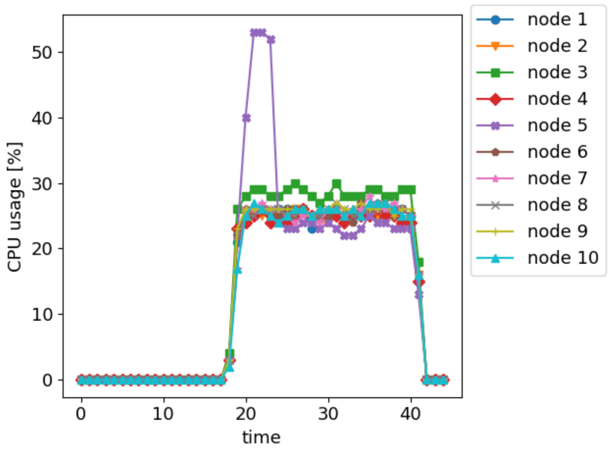
(b) clients with empty request, 4 peers



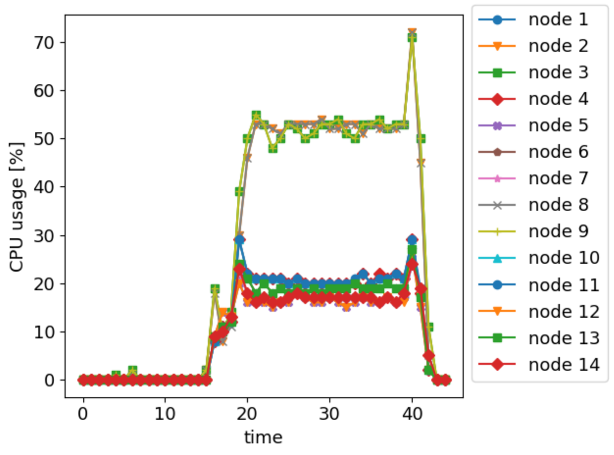
(c) servers with requests of 2048 B, 7 peers



(d) clients with requests of 2048 B, 7 peers

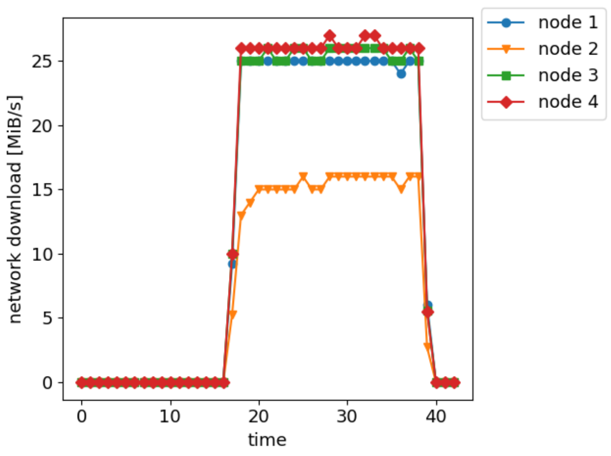


(e) servers with requests of 2048 B, 10 peers

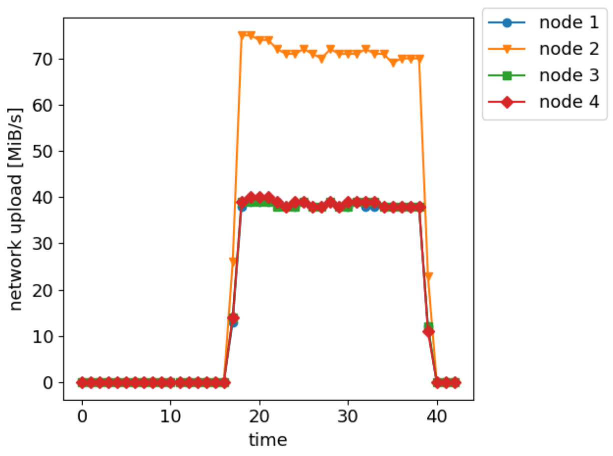


(f) clients with requests of 2048 B, 10 peers

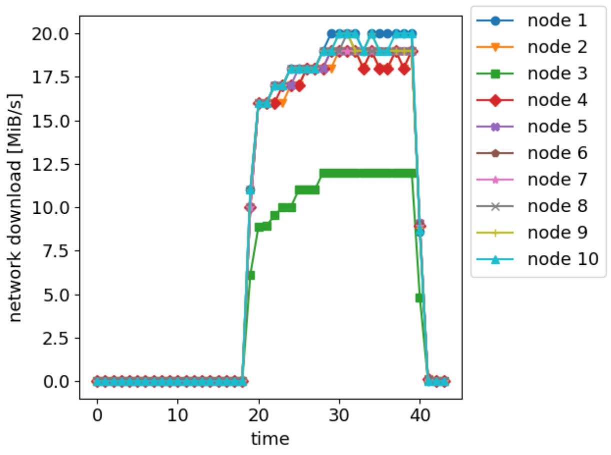
Figure 4.7: CPU usage on both the server and client side for different message sizes and server counts



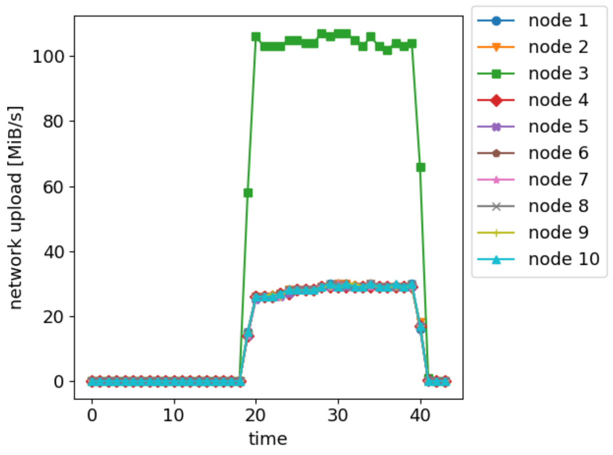
(a) download with empty requests and 4 peers



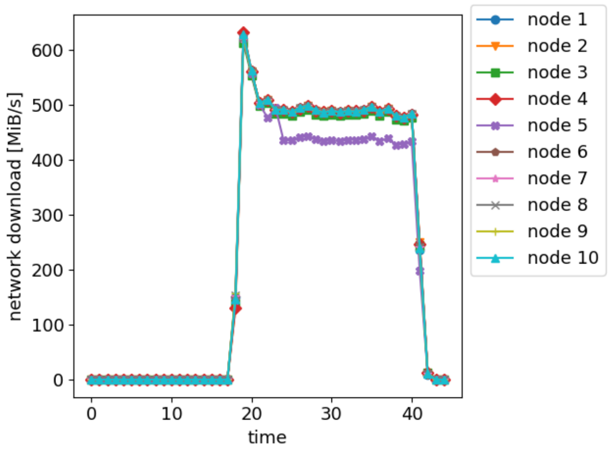
(b) upload with empty requests and 4 peers



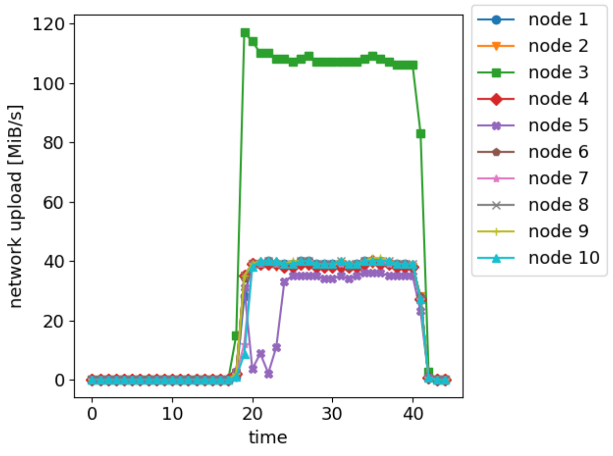
(c) download with empty requests and 10 peers



(d) upload with empty requests and 10 peers



(e) download with requests of 2048 B and 10 peers



(f) upload with requests of 2048 B and 10 peers

Figure 4.8: Hotstuff network usage.

4.3 Our implementation

In the benchmark of our implementation, we used the node counts and payload sizes common to our hotstuff and moduBFT experiments. To compare to moduBFT, we measured our performance with asynchronous factor 32 and message authentication codes. The results can be found in figure 4.9. We can see that our implementation enjoys superior performance in all the setups but the size of the performance gain depends on the number of peers. While with 4 peers, we achieve up to 4.7-times larger throughput (236 vs 50 KOps/s for payload size 512 B), with 10 peers we only get a speedup of approximately 2x for the same payload size (69 vs 35 KOps/s). Our latency follows a somewhat similar pattern — with 4 peers moduBFT reaches saturation at 10 ms latency, while we do at 6 ms, with 10 peers, moduBFT reaches its near 35 Kops/s peek throughput around 20 ms, the same as we do.

CPU utilization can be found in figure 4.10. The utilization grows with the number of peers, indicating that we succeed in obtaining parallelism with the number of peers. However, as was the case with hotstuff, the utilization is low throughout most of the experiment.[3] We will analyze our implementation's bottlenecks later.

4.3.1 Effect of asynchronous factor

We evaluated the effect of asynchronous factor on our performance. Since our hotstuff experiments were run with asynchronous factor 4200, we chose this to be our large value of the factor. The results, which can be found in figure 4.11, show significantly increased throughput at the cost of larger latency.

4.3.2 Effect of public key signatures

We implemented support for using public key signatures for all messages. This signature scheme is used by hotstuff, except hotstuff also uses batching making their performance not directly comparable. The effect of batching will be discussed later.

We measured the performance with this more demanding cryptography with asynchronous factor 32. We present the results in figure 4.12. Due to the large difference in both latency and throughput, we chose logarithmic scale and show the highest throughput observed and the corresponding latency.

4.3.3 Effect of batching

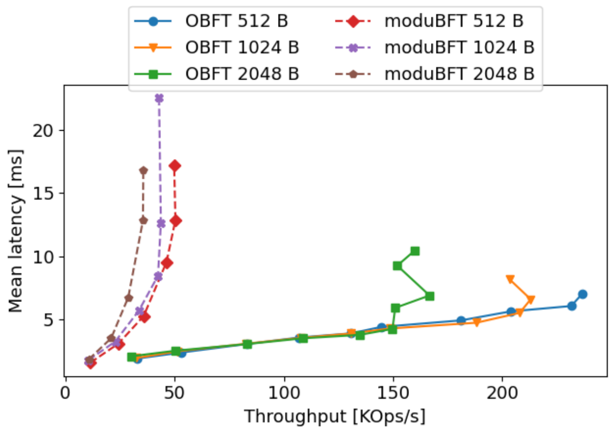
To assess its potential improvements, we used a test of our cryptographic module. This test first creates an array of messages roughly simulating those a leader of 4 peers would receive — each request with payload of 1024 B is followed by 4 prepare and 4 commit messages, in total, the array contains 10000 messages. A part of the test iterates through the array and signs and verifies the messages in a single thread before printing the throughput. To simulate batching, we simply changed the iteration logic to identify a segment of b consecutive messages and pass them to the `sign` or `verify` function as one message (by giving it the pointer to the first message and the total length). When run with a batch size of $b = 400$, we saw a speedup of 299x (approximately 45K msgs/s vs 1.5K msgs/s [4]).

From this, we concluded that the time to sign a number of messages with public key signatures is largely dependent on the message count and less so on the sum of their lengths. Thus, we could expect a significant speedup of cryptographic operations when batching is introduced.

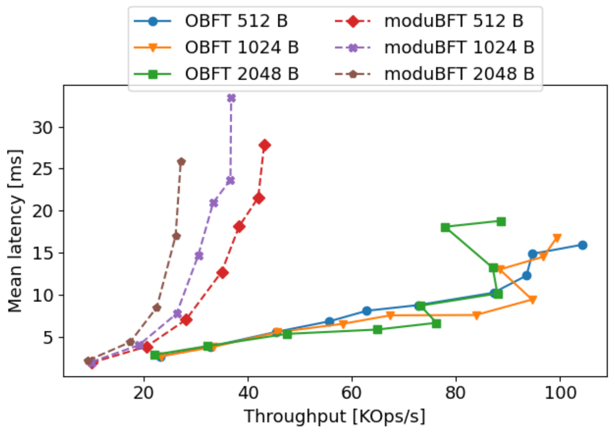
If we apply the optimistic speedup of 299x we saw in the crypto buffer test to the results with PK signatures, we obtain a theoretical throughput of approximately 350K − 470K operations per second for 4 peers. Since we have seen higher throughput when running with MACs (see figure 4.11), we expect similar throughputs may be achievable with asynchronous factor 4200 and 4 peers. With higher peer count, we have seen a performance decrease in figure 4.11, which suggests that even with large performance gains from batching, such performance may be unachievable because other parts of the program become the bottleneck.

[3]The exception is the end of the experiment. We have generally observed high CPU utilization on servers with no requests arriving. We suspect this could be caused by frequent flushing described in section 3.5

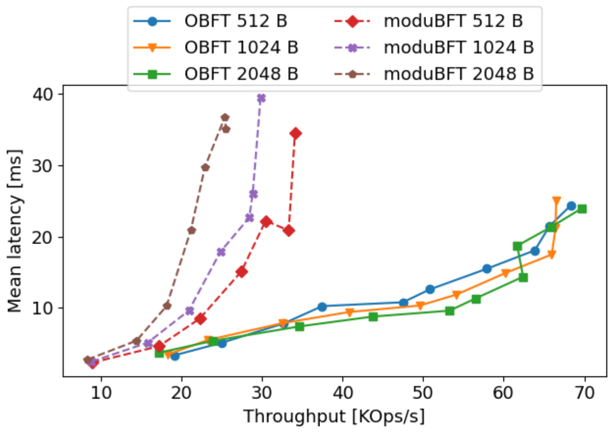
[4]to present raw data, we state we saw 454545 msgs/s vs 1517 msgs/s in one experiment



(a) 4 peers

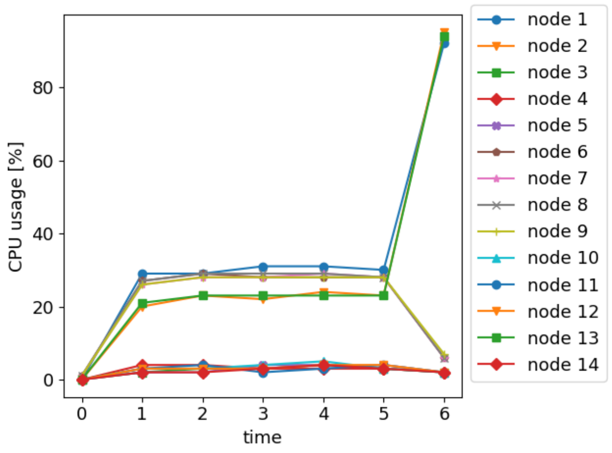


(b) 7 peers

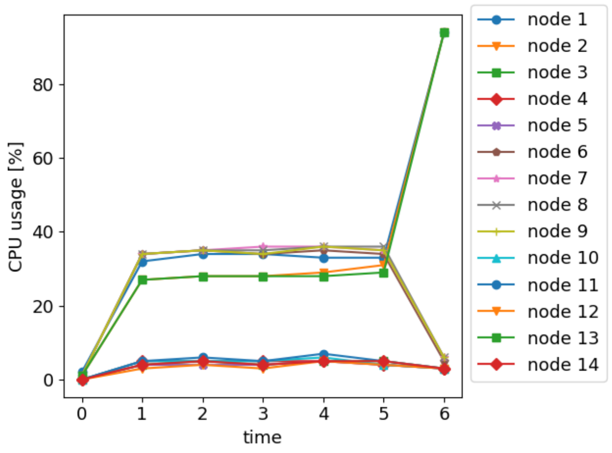


(c) 10 peers

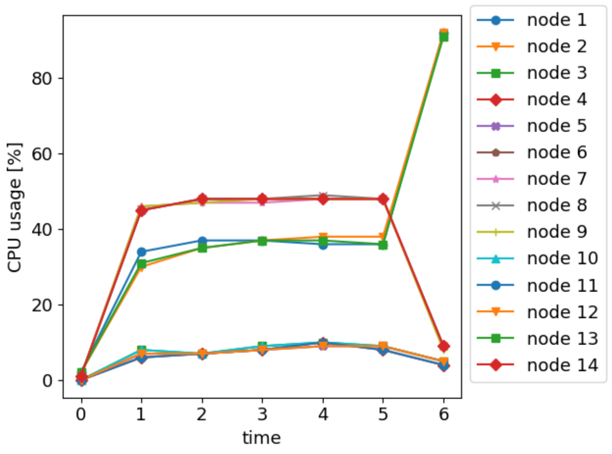
Figure 4.9: Comparison of our implementation to moduBFT. We used asynchronous factor 32 and MACs.



(a) 4 peers

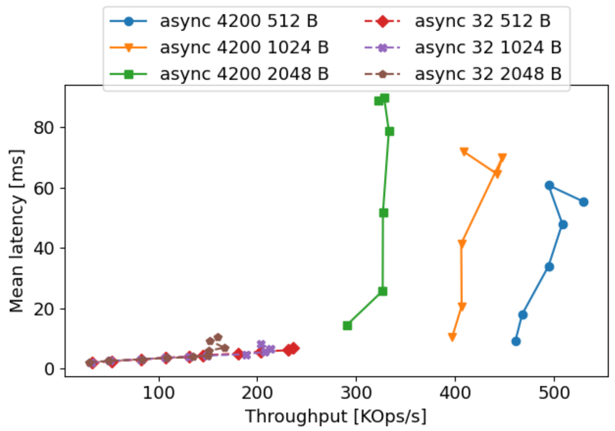


(b) 7 peers

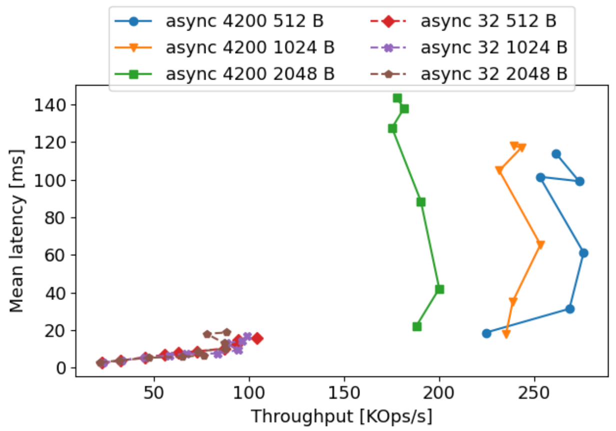


(c) 10 peers

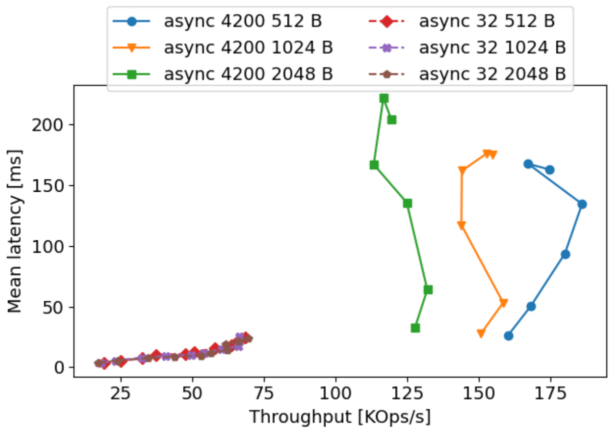
Figure 4.10: CPU usage with MACs and asynchronous factor 32.



(a) 4 peers

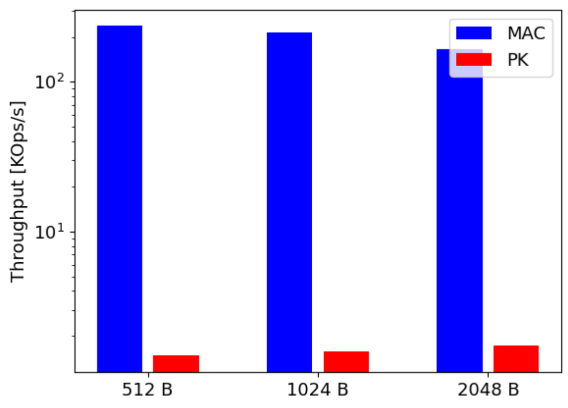


(b) 7 peers

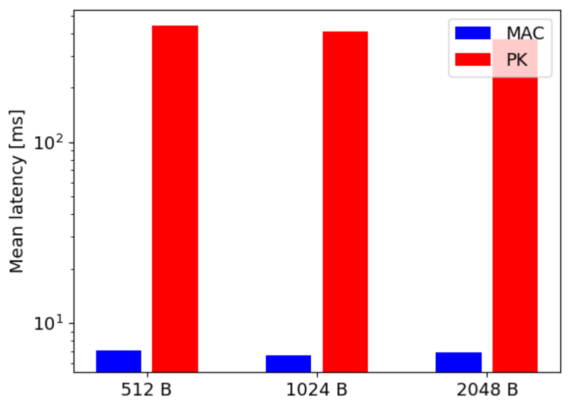


(c) 10 peers

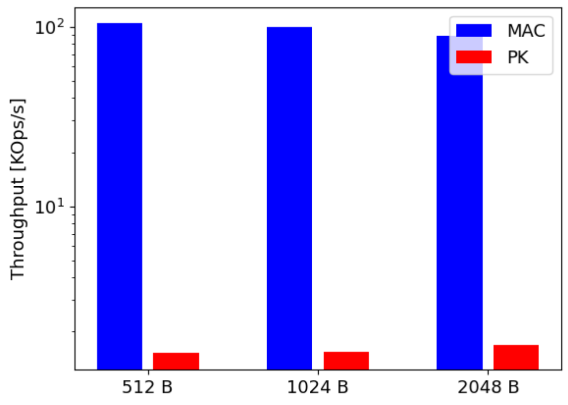
Figure 4.11: Comparison of our implementation with asynchronous factor 4200 to our normal setup.



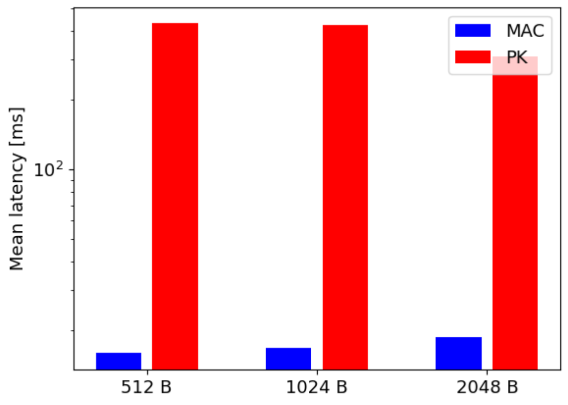
(a) 4 peers throughput



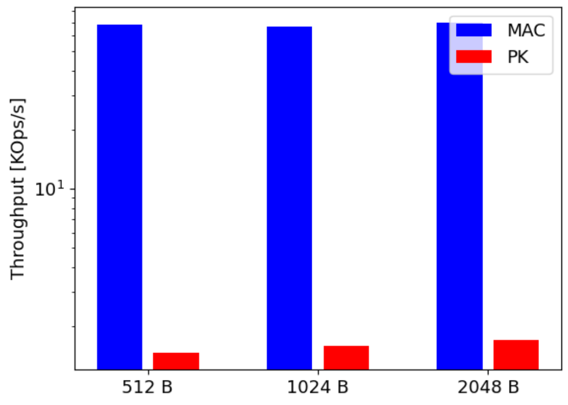
(b) 4 peers latency



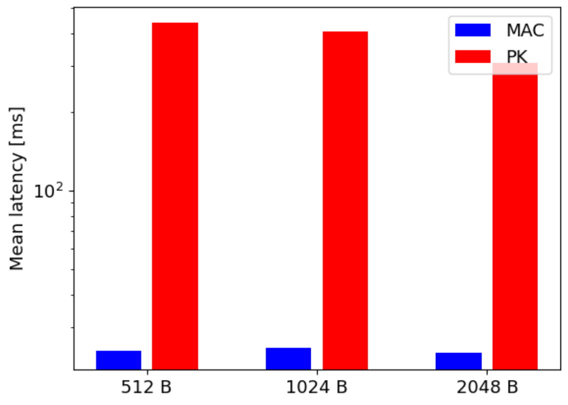
(c) 7 peers throughput



(d) 7 peers latency



(e) 10 peers throughput



(f) 10 peers latency

Figure 4.12: Comparison of using PK signatures for all messages to using MACs. Presented is the highest observed throughput and the corresponding latency. Note that the y-axis has log scale.

4.3.4 Bottleneck analysis

To help assess which parts of our implementation would be suitable for further optimizations, we explored which parts formed significant bottlenecks.

Cryptographic module When public key signatures are used, the cryptographic module becomes a significant bottleneck, as evidenced by comparing this setup to one with MACs (see figure 4.12). However, does this still hold when MACs are used? To measure the maximum throughput of the cryptographic module, we used our test with simulated messages as described in section 4.3.3. We used one cryptographic buffer for signing the messages and another one for verifying the signed ones. With 4 threads used for signatures using MACs and 4 for verification, we were able to obtain a throughput around 3.8M messages per second [5]. With 4 peers, and thus 1 request, 3 prepare and 3 commit messages received on each node per request, this translates to approximately 540 KOps/s. And with 10 peers, and thus 1 request, 9 prepare and 9 commit messages per request, this translates to approximately 200 KOps/s. Both are much higher than the respective throughput we have seen.

We also compared this with signing and verifying all messages in a single thread without using the cryptographic buffer and instead calling the `sign` and `verify` functions directly. This single-threaded version achieved a throughput of approximately 1M messages per second [6]. This means that with 4 signing and 4 verifying threads, we achieved a speedup of 3.6x. When the number of threads was increased to 8 for signing and 8 for verifying, we experienced throughput of 5.1M messages per second [7], corresponding to speedup of 4.8x. This leads us to believe that if needed, the throughput of our cryptographic module could potentially be increased further by using more threads.

These two facts suggest that the cryptographic module doesn't form a significant bottleneck when MACs are used.

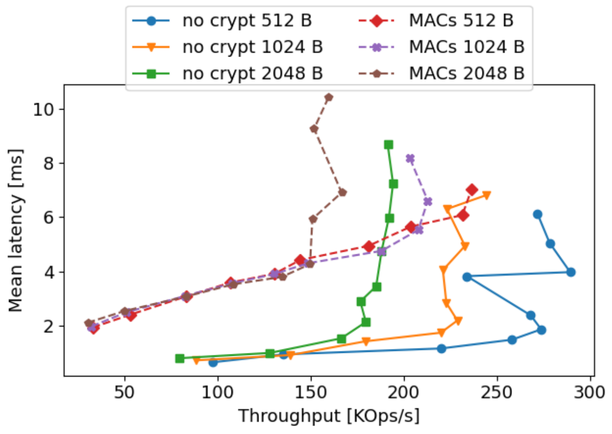
Lastly, we performed experiments running the rest of the PBFT protocol, without cryptographic operations but with the same message structure, including the appropriate number of bytes that would normally be used by the authenticator. The results are located in figure 4.13. We can see that for higher peer counts, the throughput is similar to running the entire protocol with message authentication codes. This is evidence that, at least for higher peer counts, our implementation is likely bottlenecked by either the networking, or the state-machine module. On the other hand, the slimmed-down version exhibits much lower latency, meaning that potentially significant latency improvements may be possible to gain by offloading cryptography.

Networking In order to evaluate whether the program is bottlenecked by networking, we inspected the message rate and measured bandwidth in our main experiments. We present the bandwidth in figure 4.14. We will now analyze the relationship of throughput (measured in requests per second), message rate on a single server and network download bandwidth used. Consider there are p peers. For every request, the state-machine module of a single node handles 1 message with the request, $p - 1$ prepare messages and p commit messages, $2p$ messages total. However, one of the commit messages comes from itself and therefore doesn't go through the network in our implementation, making the network module handle $2p - 1$ messages per request. Each prepare and commit message contains a 64-byte header and p MACs each consisting of 32 bytes, which makes them $64 + 32 * p$ bytes long. Considering a payload size of s, each request also contains s bytes of payload making it $64 + 32 * p + s$ bytes in size. Therefore, with a throughput of t requests per second, we would expect a download of approximately $((64 + 32 * p) * (2p - 1) + s) * t$ bytes per second on each node. Table 4.1 lists the computed message rates and bandwidth for some of our saturated experiments. The bandwidth approximately matches our measurements and we can see that it remains approximately the same even as the throughput decreases to less than a third as the peer count grows. The message rate, on the other hand, decreases almost by half of its initial value. Since we saw the message rate decreasing significantly without larger changes in the bandwidth, we suspected the networking module fails to provide enough messages to the state-machine to saturate it, and thus that networking (possibly our networking module) likely forms a bottleneck of our implementation.

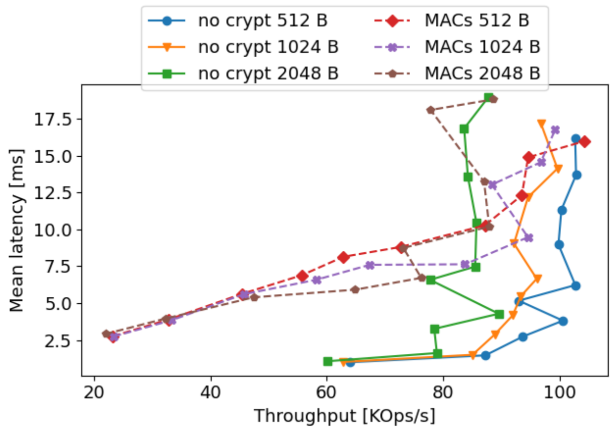
[5] 3816793 msgs/s in one experiment

[6] 1054852 in one experiment

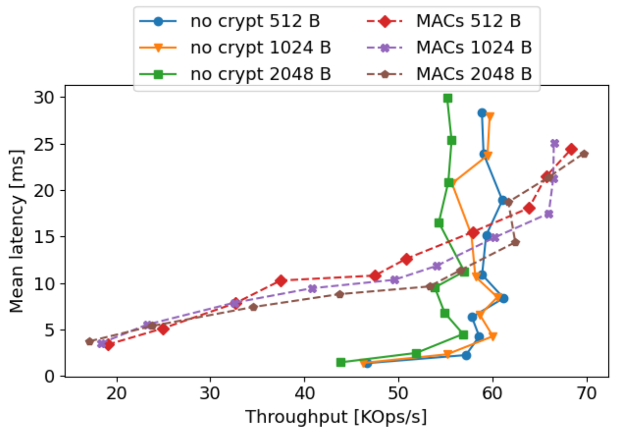
[7] 5154639 in one experiment



(a) 4 peers



(b) 7 peers

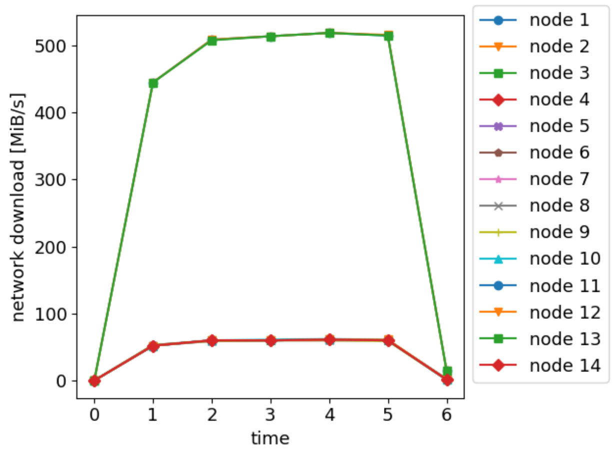


(c) 10 peers

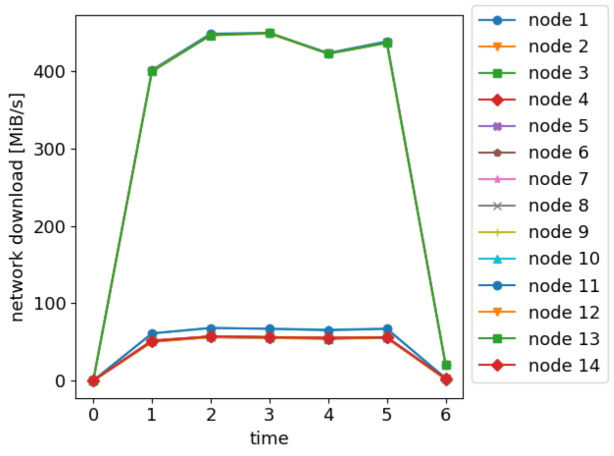
Figure 4.13: Comparison of our implementation with and without cryptographic operations.

#peers	tput [KOps/s]	msg rate [KMsgs/s]	bandwidth [MiB/s]
4	272	1 904	481
7	110	1 326	414
10	70	1 121	439

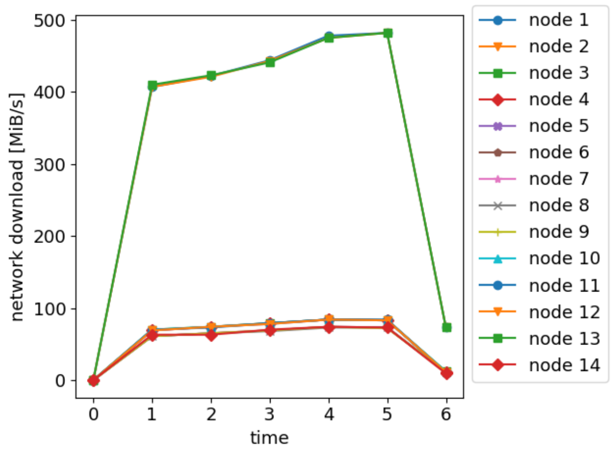
Table 4.1: Computed message rate and network bandwidth for different peer counts and observed throughputs with payload size of 512 B with no cryptography and 52 peers.



(a) 4 peers



(b) 7 peers



(c) 10 peers

Figure 4.14: Network download on peers (first p nodes) and clients with payload 512 and different peer counts with no cryptography used.

#peers	tput [KOps/s]	msg rate [KMsgs/s]	SMM max msg rate [KMsgs/s]
4	272	1 904	23 696
7	110	1 326	16 778
10	70	1 121	11 013

Table 4.2: Comparison of message rates computed based on measured protocol throughput and measured maximum message rate attainable by the state-machine module (SMM) alone.

State-machine To verify the above hypothesis that networking forms a bottleneck in our implementation, we tested the maximum message rate of the state-machine module. We used similar messages to those from cryptographic module test and called the `handle_msg()` function on each. This function implements the processing of a single message by the state-machine module. The results, presented in table 4.2, show that the the state-machine module is capable of handling significantly higher message rate for all peer counts, supporting our hypothesis. Interestingly the maximum state module message rate also decreases with growing number of peers. A possible reason could be the increase in the message size (caused by larger authenticators), resulting in more memory movement when storing messages to the message log.

Chapter 5

Future work

5.1 Cryptographic batching

We were unable to directly compare to hotstuff because of the fact that hotstuff supports signing a number of messages (400 by default) with a single signature, while our implementation signs each message individually. Our experiments with this kind of batching (presented in chapter 4) indicate that significant increases in throughput can be gained this way. We started implementing batching support but were unable to finish this task in the time available. We will now describe some of the challenges faced as well as two possible design.

The first approach would be to perform batching almost transparently in the cryptographic module. The state-machine module would send the same messages and the cryptographic module would assemble them into batches before creating their signature. However, in our implementation, messages are signed in a cryptographic buffer, which is common to all messages regardless of their recipient. And messages in a single batch (and thus sharing one signature) should have the same recipient as the entire batch is required to verify the signature. Having different recipients in a batch would require sending the batch to all recipients (including potentially the clients), which would be wasteful. To avoid this, we propose grouping messages by their recipient before passing them to the cryptographic buffer. This requires having an additional buffer for each recipient to store them, creating more memory movement but the performance gains from batching would likely overshadow this additional cost. A message being sent would be placed in the appropriate batch. When a batch would contain a sufficient number of messages, the entire batch would be wrapped in a single message of a special type *batch*, signed and sent out. When receiving the batch, a node would extract its messages and handle them one-by-one.

An alternative would be assembling a batch in the state-machine module. A number of incoming requests could be assembled into a single pre-prepare message subsequent prepare and commit messages would vote for all the assembled requests at once. This would have the added benefit of reducing the number of messages sent because each replica could send a single prepare message for each batch. This would reduce the load on networking as well as possibly making view-changes easier than the previous technique (where one batch could include messages valid for the view change along with some that are not). Therefore, we consider this approach to be preferable, although it may require larger changes to the state-machine module.

5.2 Hardware offloading

We developed this work as a platform for experimenting with hardware offloads, particularly FPGAs. This makes these offloads an obvious item for future work. We see potential for the following offloading strategies, likely each building on top of the previous ones. We have started integrating our implementation to a broadcast offloading implementation available in our research group [22] but this is still very much work in progress.[1]

[1] At the time of writing, it can be successfully compiled and parts of it can be run but still it needs to be debugged to be able to complete all phases and provide answers to the clients. Afterwards, it should be optimized for performance.

5.2.1 Networking

A common communication pattern in the protocol is sending a single message to many recipients (namely all $n$servers) — this is the case with prepare, commit and checkpoint messages. This is currently done by copying the message n times and passing it to the n threads who call `send()`. Alternatively, this could be done in hardware — the message would not be copied in software and instead, a single copy could be passed to an FPGA to and sent to multiple recipients, removing the copying and performing the TCP protocol in hardware.

This could be achieved by changing the implementation of the `net_enq_msgs()` function in the networking module. This function gets a pointer to a buffer containing messages, its length and the intended recipient. In the current implementation, it performs the copying of the message and placing it in the networking buffer. Instead, it could detect whether the message sending should be offloaded and issue a command to the FPGA if appropriate. We expect the offloading to be done only for communication among the peers since this is where the protocol calls for a broadcast.

We have an at our disposal an existing implementation of such broadcast [22], which uses the Coyote FPGA driver [30]. We have started working towards using this broadcast implementation in our program. It already uses a header, which has similar structure to ours, and a payload. We will now describe the differences and how they can be handled as well as other details that would require attention when implementing the offloading.

- **Length entry** The length specified in our header includes the length of the header itself, while the broadcast offload expects it to contain only the payload size. This could be handled by subtracting the header length before sending the message and readding it upon receiving it. Since the length entry is used in other modules, this would have to take place in the networking module, likely in `net_enq_msgs()` and `net_get_msgs()` functions.

- **Missing and misplaced entries** Unlike the broadcast, our implementation has client id and timestamp entries. These would have to be added. Since the broadcast header is padded to 64 bytes, this should not pose significant problems. Also, the order of our header entries is not necessarily the same as that used in the broadcast header. This can be also be changed when changing the length.

- **Digest** Our `struct msg` contains the request (or checkpoint) digest, unlike that of broadcast implementation. However, this additional entry could become a part of the message payload instead of being part of the header.

- **Recipient** When performing broadcasting to a number of recipients, the FPGA would change the recipient entry in each message copy from the special value of -1 to the single recipient. This would make the signature invalid because the entire header (including the recipient entry) is part of the signed bytes. To mitigate this problem, the software on the receiving side would have to detect that the message was sent by an offloaded broadcast and restore the original recipient entry of -1. The detection could be performed based on the message type — for example, we could offload the broadcast of all prepare, commit and checkpoint messages and no other.

- **Padding** The broadcast header is padded to 64 bytes, while our header (excluding the digest) is smaller. Since the header needs to be immediately followed by the payload, this requires either copying the payload into a segment of memory large enough to accommodate this larger header, or padding our header to the same size. The latter requires less memory movement when an FPGA is used and is easy to implement but increases message size, which may slow down the software-only reference implementation. Therefore, if this approach is taken and compared to the software-only implementation, this reference should use our original, unpadded header to be a fair comparison.

5.2.2 Cryptography

Our experiments indicate that when public key signatures are used, they become the bottleneck of the protocol. Even if message authentication codes are used, and thus large throughput obtained, we have observed that cryptographic operations have significant impact on latency. This makes cryptography a good candidate for offloading.

We expect that cryptography would be offloaded only after networking offload was implemented. Messages would be processed by the state-machine module and passed to the FPGA, where they would be signed and immediately thereafter sent without being passed back to software. On the receiving side, the software would receive the verified messages. The cryptographic module would by bypassed in a similar way to our support for running without cryptography, which is already in place. Since our state-machine module needs to receive signatures of some messages[2], the accelerator would have to return the signature along with the message.

5.2.3 Counting

A final hardware offloading option would be the counting of prepare, commit and checkpoint votes. The previous two approaches suggested handling the broadcasting, signing and authentication of messages to be performed by hardware but every received message would eventually be passed to software. However, in many cases, most of what the software would do would be simple counter incrementation. An alternative would be counting the number of votes received in hardware and only getting the software involved if the counter receives the applicable threshold — for instance if $2f$ prepare messages have been received.

Implementing this would require changes to the core of the state-machine module, the `handle_msg()` function. The current implementation expects to receive every message and performs the counting. It also calls itself recursively when the node is sending messages to itself. This is done to avoid verifying the message and having to copy it through all the applicable receiving buffers. Therefore, either the software would have to be able to instruct the FPGA to increment its counter, or any message a node sends to itself would have to go through the normal receiving path that messages from other nodes take.

5.3 Further software optimizations

In this section, we propose possible exploration directions that could be taken to try to improve our performance using only software. Previous analysis indicates that our implementation may be bottlenecked by networking, making this module an obvious choice for optimizations.

Many of the messages being sent are intended for all peers. When such a message is enqueued, it is copied to all peer output buffers. This copying is done in a single thread and could potentially bring performance penalties. An alternative would be introducing a separate buffer for messages to all peers and keeping track of which recipients it has been sent to. This could be done by each thread having an extra set of start and end pointers as described in section 3.2.1. This extra set would belong to the new buffer. When a message for all recipients would be enqueued, it would only need to be placed in the single common buffer and all `local_end` pointers would be increased. To make sure no valid bytes are overwritten during this operation, each of the new `local_start` pointers would have to be checked before the enqueue.

Next, our MACs are still larger than they could be, making prepare and commit messages (which are broadcast by each node to each other node) significantly larger. The PBFT paper [11] suggests truncating the hash of the message to 10 bytes, while we still use the entire 32-byte hash.

Furthermore, in section 3.2.1, we mention that to reduce the number of system calls required, we allocate larger network buffers and copy bytes into a contiguous segment of memory before sending. This optimization was introduced early in the development and we have increased our buffer size multiple times since then. This increase in size makes the byte copying more expensive. Therefore this optimization should be reevaluated.

Lastly, our cryptographic buffers have been designed and tuned with MACs. Perhaps higher performance with public key signatures could be achieved by tuning parameters such as the chunk length.

5.4 SIMD

We also wanted to explore optimizations through the use of SIMD instructions. These instructions perform the same operation on multiple pieces of data at once, often with the same latency and throughput as those needed for processing a single piece of data. However, we only saw small potential for their use in our PBFT implementation.

[2]It needs signatures of resubmitted requests to be able to forward them as well as signatures for pre-prepare and prepare messages, which need to be stored in the node's log for possible later view-changes.

The bulk of the state-machine module, presented by the decision-making function `handle_msg()` performs rather heterogeneous operations, which does not make it a good fit. Messages of type request and pre-prepare are handled differently from prepare or commit messages as there is no vote counting on the former. The counting would have, upon first glance, some potential for use of SIMD but the counters to be incremented by different messages are not necessarily consecutive in memory making the storing of their values difficult. Furthermore, if a batch of messages is taken for SIMD processing, it may contain two votes for the same counter, which should be increased by two. This poses a further challenge for SIMD. Lastly, we do not believe this function is a significant bottleneck.

We imagine SIMD could be used as a minor optimization for doing sanity-checks. For example, replicas are to refuse messages with request ids below a certain low water mark or above a certain high water mark. However, we do not expect this would have a large performance impact.

The networking module performs hardly any calculations at all, it only moves the message to the appropriate buffers, performs communication between threads and sends it. The current implementation copies each message for n recipients n ways. However, we expect that the implementation of `memcpy` already makes use of nearly all the load and store capacity available making optimizations of this copying implausible.

The last part of the program is cryptography, which have entrusted to a library making it impossible to make any low-level optimizations. The library only takes one message at a time for signing or verification. Given that our results indicate the time to sign k messages using a public key is largely dependent on the number k and less on the sum of their lengths, we expect SIMD could in principle be used to sign (or verify) a number of messages at once if public keys are used. However, such an approach would require manual implementation of asymmetric cryptography, which is outside the scope of this thesis.

5.5 Speculative digest checking

Our implementation of the speculative digest checking described in section 3.3.4 is incomplete. It handles all normal operations but if one of the received digests is incorrect, the replica needs to zero its counters and replay received messages from its logs and this part hasn't been implemented. When we receive prepare or commit message, we check whether the pre-prepare has arrived. If so, we check the digest as usual. If not, we store the digest of the received message. What should be done in the latter case is to find out whether this prepare is the first received message for its request id and based on this information either store it or check it against the previously stored one. Finishing our implementation would require maintaining a set of boolean flags to make this detection and implementing the replay of received messages (which are available in the logs) in case one of the digests is wrong.

5.6 State-machine replication

In our implementation, replicas reach a consensus in establishing a total order of requests from clients. However, the original PBFT paper [11] proposes using this protocol for state-machine replication. In this setting, there is an underlying service, which is replicated to all servers and whose state can be observed and modified by client requests. Namely, the paper uses NFS as their underlying service. This section describes how our implementation could be extended to serve as a library for byzantine fault tolerant state-machine replication of arbitrary underlying service.

An underlying service could be supported by making the following changes. On the server side, the service state could be implemented by storing a pointer (of type `void *`) to it. This pointer could be stored in our structure maintaining the protocol state (`struct node`) and set by a user-supplied initialization function. An additional cleanup function could be added to free any space allocated by the initialization. Another user-supplied function could perform request execution given a pointer to the request body and the service state. It would be called when committing a request and generate the reply to send to the client. Furthermore, the state of the service at checkpoints should be kept, requiring the user to supply one more function for this checkpoint creation.

On the client side, similar callbacks could be exposed to generate the next request and process a received reply. Alternatively, to obtain a more convenient interface (at the cost of larger implementation effort), the user code could be made the caller, using our implementation as a library. Functions to initialize and tear down connections, send a request and wait for a reply could be exposed. The last named would count the number of replies, check whether they match and return the reply received from the servers as well as performing any resubmits if needed.

These changes would allow for performing a service during normal operations. The next section describes the changes needed for correct view changes with an underlying service.

5.7 Re-executing requests after a view-change

We implemented a complete detection that a view change is necessary and the complete scheme of voting for it. This includes the content of view-change messages, which carry a proof of the last stable checkpoint, a list of all requests that finished the prepare phase on the voting node and the pre-prepare and prepare messages proving this fact.[3] However, we have each follower restart normal operations immediately after receiving and verifying the new-view message, whereas the protocol calls for reverting to the last stable checkpoint and re-executing the prepared requests. The reverting could be implemented by resetting the protocol state (e.g. setting prepare vote counters of all requests since the checkpoint to zero) and letting the user supply an additional callback to reset the underlying service state.

Furthermore, some replicas may not be aware of the last checkpoint. The protocol doesn't guarantee that when a replica has reverted to the checkpoint and starts re-executing requests, it has received the body of all the requests it needs to re-execute. If the state of some underlying service is to be kept, this information needs to be available to continue execution. The authors of PBFT call for the replica to fetch this information from its peers who have sent their prepare vote for the request. If the replica is not aware of the checkpoint that it should revert to, it should also fetch this information from others. Implementing this would require addition and handling of new message types that represent these queries and their answers as well as implementing support for querying of the underlying service state.

Finally, our new-view messages are missing the pre-prepare messages generated by the leader in the new view, which would be needed in subsequent view changes if reverting to a checkpoint and re-executing of requests is implemented. Sending these in the new-view message would increase its size, which would require increasing the size of our networking buffers to be able to accommodate such a message. An alternative could be the leader sending these pre-prepare messages separately and the followers only re-executing requests when they have received the new pre-prepare message. Our communication with ordered reliable message delivery should allow for this.

5.8 Robustness

Our implementation focuses on performance during normal operations and therefore its resilience to attacks was not extensively tested. If our implementation were ever to be used in industry setting, this testing would have to be done and the implementation may have to be made more robust to receiving invalid messages. This section briefly mentions some possible attacks we are aware of and how we handle them.

Duplicate votes In the situation when replicas are sending checkpoint messages, the votes are counted towards the goal of $2f + 1$ messages. A single faulty replica could send multiple messages for the same checkpoint to convince its peers to accept the checkpoint prematurely or even accept an invalid checkpoint (one that no honest replica has voted for). If replicas were susceptible, they could be convinced to accept so many invalid checkpoints that they would discard valid ones. A similar potential problem is present when counting prepare and commit votes. To mitigate this problem, each of our replicas maintains information about which votes it already receives and ignores duplicates. We do this for votes received during normal operations but not for view-change messages or the prepare messages contained in them. Furthermore, although we have seen this safeguard rejecting inadvertent duplicate votes, we have never intentionally attacked it, therefore more testing should be done for real use.

[3] See section 2.2.1 for details of the view-change message structure

Malformed messages Faulty replicas can send messages with invalid entries in the header. For instance, the sender field could be set negative or higher than the number of nodes. Processing such a message has potential for array out-of-bounds accesses and memory corruption. We attempt to sanity-check all received messages and ignore invalid ones but no testing was done.

Invalid message length One particular entry in message header that could be set to a wrong value is the message length. This way, faulty nodes could attempt to corrupt later messages — for instance if the length was set too high, the header of the following message would be interpreted as part of the attacking message's payload. However, our networking module reads the message length and always returns the appropriate number of bytes. And since this module handles messages separately for different connection, all messages a faulty node is able to corrupt are its own.

Too large messages Our buffers have a finite size and thus there is an upper bound on the message size our nodes are able to accept. To some extent, this is a general problem as the nodes will always have only finite memory available. Proper handling of unrealistically large messages should be implemented (likely ignoring them and either regarding the node as faulty or skipping to its next message).

Network unresponsiveness We use networking with reliable message delivery, which requires the other side of the connection to confirm its receipt. If this never happens, for instance because the node is down, the `send()` system call never terminates, which leads to the respective send buffer to get full (see section 3.2.3). Handling of this situation should be implemented. Given the finite memory available on peers, this would likely involve dropping some of the messages, thus violating reliable message delivery. This would in turn require handling additional corner cases that absence of reliable delivery would allow (as described in section 3.4).

Chapter 6

Conclusion

In this work, we present OBFT, a high-performance modular Byzantine Fault Tolerant consensus implementation in C, which paves the road towards exploration of hardware offloading of the PBFT protocol. We show that compared to moduBFT, a previous work exploring potential gains of future offloading, we reach up to 4.7 times higher throughput at equivalent or smaller latency.

To aid future offloading, we have divided the implementation into different modules, responsible for networking, cryptography and keeping the protocol state respectively. This allows for measuring performance of individual modules as well as aiding in future offloading. For instance, if cryptographic operations are to be offloaded, the cryptographic module can be bypassed.

We have analyzed bottlenecks present within the current implementation to asses the potential of different offloading strategies. Our experiments indicate that when public key signatures are used, they become a significant bottleneck unless large batches were to be used. This justifies offloading of cryptography. When message authentication codes are used, the implementation seems to instead be bottlenecked by the networking part, although cryptography is still responsible for increased latency.

With this information in mind, we have discussed three methods of offloading to FPGAs. Firstly, the network broadcast pattern present in the protocol can be performed in hardware, possibly improving networking performance. Secondly, once this is done, the cryptographic module can be bypassed — instead, signatures can be computed by the FPGA once it receives the messages for broadcasting, and verified before passing them back to software. This targets the two main bottlenecks we have identified. We have described in more detail, how these changes can be made in our implementation. Lastly, a third offloading method is feasible, which would implement the counting of prepare and commit messages in hardware, which could boost performance of the state-machine module.

Bibliography

[1] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance, 2017.

[2] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization, 2022.

[3] Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho, Michael Ferdman, Peter Milder, and Shuai Mu. Waverunner: An elegant approach to hardware acceleration of state machine replication. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 357–374, Boston, MA, April 2023. USENIX Association.

[4] Amazon. Amazon Managed Blockchain. https://aws.amazon.com/managed-blockchain/. Accessed: 17.8.2023.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, apr 2018.

[6] Balaji Arun and Binoy Ravindran. Scalable byzantine fault tolerance via partial decentralization. *Proc. VLDB Endow.*, 15(9):1739–1752, may 2022.

[7] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[8] Saman Biookaghazadeh, Pravin Kumar Ravi, and Ming Zhao. Toward multi-fpga acceleration of the neural networks. *J. Emerg. Technol. Comput. Syst.*, 17(2), apr 2021.

[9] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. 2016.

[10] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget, 2019.

[11] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. *OSDI*, 03 1999.

[12] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[13] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. Hardware acceleration of compression and encryption in sap hana. *Proc. VLDB Endow.*, 15(12):3277–3291, aug 2022.

[14] Evangelos Deirmentzoglou, Georgios Papakyriakopoulos, and Constantinos Patsakis. A survey on long-range attacks for proof of stake protocols. *IEEE Access*, 7:28712–28725, 2019.

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

[16] ETH systems group. Heterogeneous Accelerated Compute Cluster. https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc.html. Accessed: 14.8.2023.

[17] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, apr 1985.

[18] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An open-source 100-gbps nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 38–46, 2020.

[19] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Vincent Gramoli. From blockchain consensus back to byzantine consensus. *Future Generation Computer Systems*, 107, 09 2017.

[21] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure, 2019.

[22] Zhenhao He. Example benchmarking of bft. https://github.com/zhenhaohe/Coyote/tree/bft/sw/examples/bft. Accessed: 7.8.2023.

[23] Zhenhao He, Dario Korolija, and Gustavo Alonso. Easynet: 100 gbps network for hls. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 197–203, 2021.

[24] Kexin Hu, Zhenfeng Zhang, Kaiwen Guo, Weiyu Jiang, Xiaoman Li, and Jiang Han. An optimisation for a two-round good-case latency protocol. *IET Information Security*, 17(4):664–680, jul 2023.

[25] IBM. IBM Blockchain. https://www.ibm.com/blockchain. Accessed: 17.8.2023.

[26] IBM. Learn step-by-step how to set up a basic blockchain network. https://developer.ibm.com/learningpaths/get-started-blockchain/use-ibm-blockchain-platform/. Accessed: 17.8.2023.

[27] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 425–438, USA, 2016. USENIX Association.

[28] Jeff Barr. AQUA (Advanced Query Accelerator) – A Speed Boost for Your Amazon Redshift Queries. https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/. Accessed: 15.8.2023.

[29] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256, 2011.

[30] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on fpgas? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010. USENIX Association, 2020.

[31] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. volume 51, pages 45–58, 01 2007.

[32] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[33] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, jul 1982.

[34] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3:202–215, 2005.

[35] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[36] NVIDIA. NVIDIA Mellanox ConnectX-5 Adapters. https://www.nvidia.com/en-us/networking/ethernet/connectx-5/. Accessed: 15.8.2023.

[37] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[38] OpenSSL. OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/. Accessed: 14.8.2023.

[39] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, apr 1980.

[40] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 13–24. IEEE Press, June 2014. Selected as an IEEE Micro TopPick.

[41] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olya Ohrimenko, Felix Schuster, Roy Schwartz, Alex Shamis, Olga Vrousgou, and Christoph M. Wintersteiger. Ccf: A framework for building confidential verifiable replicated services. Technical Report MSR-TR-2019-16, Microsoft, April 2019.

[42] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43, 2015.

[43] Man-Kit Sit, Manuel Bravo, and Zsolt István. An experimental framework for improving the performance of bft consensus for future permissioned blockchains. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, DEBS '21, page 55–65, New York, NY, USA, 2021. Association for Computing Machinery.

[44] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In Gadi Taubenfeld, editor, *Distributed Computing*, pages 438–450, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[45] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput BFT for blockchains. *CoRR*, abs/1906.05552, 2019.

[46] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State-machine replication scalability made simple (extended version), 2022.

[47] The Diem Team. Diembft v4: State machine replication in the diem blockchain. 2021.

[48] Daniel Davis Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[49] Qingyang Yi, Heming Sun, and Masahiro Fujita. Fpga based accelerator for neural networks computation with flexible pipelining, 2021.

[50] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain, 2019.

[51] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, August 2016.