Improved Practical Byzantine Fault Tolerance for Blockchains

Divya Mary (u6479594)

A report submitted for the course
COMP8755 Individual Computing Project
Supervised by: Dr. Uwe Zimmer, Loong Wang
The Australian National University

May 2019

© Divya Mary 2019
Except where otherwise indicated, this report is my own original work.

Divya Mary
31 May 2019
Acknowledgments

To my supervisor, Dr. Uwe Zimmer, thank you for the patient guidance, encouragement and advice you have provided throughout my thesis.

To Loong Wang, my supervisor, my boss and above all, my first Australian "acquaintance". Thank you for being a great teacher and mentor. This thesis would not have been possible without your guidance. It would be impossible to count all the ways that you have helped me with this thesis and in my career. And for that I will be forever grateful.

To Abu, my partner, for your continued support and encouragement. Thank you for believing in me when I didn’t in myself. To my mother, father and sisters who experienced all of the ups and downs of my research.

To Maggi, my friend, thank you for all the support you have provided and for proof-reading my thesis. I consider myself lucky to have met you.

Finally, my colleagues at Ren, thank you for motivating and reminding me to work on this thesis and complete it on time. Special thanks to Ross and Vince for proof-reading this thesis and providing their valuable feedback.
Abstract

In a truly decentralized system, a consensus algorithm allows participants to communicate with each other without having to trust anyone in the process. As blockchains are a decentralized and distributed ledger, consensus algorithms form the core of a blockchain system. Majority of blockchains in today’s world uses Proof of Work (PoW) for consensus. This is the most trusted consensus algorithm in the blockchain space because it is the first consensus algorithm used for blockchains and thus has stood the test of time. However, PoW is known to have many cons, namely over-utilization of energy. Due to this, the search for cleaner and cheaper consensus algorithms had started with an aim to replace PoW. Practical Byzantine Fault Tolerance (PBFT), introduced in 1999, is the first algorithm that was proposed for achieving consensus in real-world asynchronous distributed systems with byzantine faults. However, it has not been extensively used in real-world systems due to the difficulty to scale PBFT in larger networks. This is primarily because PBFT requires a substantial amount of communication within the network in order to carry out the protocol. In order to reduce network loads, this thesis proposes an improved variation of the Practical Byzantine Fault Tolerance algorithm which pipelines and reduces the number of messages required to finalize a block in the blockchain. We found that the proposed algorithm significantly increases the throughput of blocks generated by the network.
Contents

Acknowledgments iii
Abstract v

1 Introduction 1
  1.1 Motivation .................................................. 1
  1.2 Problem Statement ........................................... 1
  1.3 Contribution ................................................. 2
  1.4 Report Outline .............................................. 3

2 Background 5
  2.1 Blockchain .................................................... 5
  2.2 Byzantine Fault Tolerance ................................... 7
    2.2.1 Byzantine Generals’ Problem ....................... 7
    2.2.2 Byzantine Fault Tolerance ......................... 8
  2.3 Consensus Algorithms ....................................... 9
    2.3.1 Proof of Work ....................................... 9
    2.3.2 Proof of Stake ................................... 10

3 Practical Byzantine Fault Tolerance 13
  3.1 Practical Byzantine Fault Tolerance ....................... 13
    3.1.1 Algorithm ........................................ 13
    3.1.2 Advantages and Disadvantages .................... 15
  3.2 Tendermint Core ............................................ 15
    3.2.1 Main Differences from PBFT ....................... 15
    3.2.2 Algorithm ....................................... 16

4 Design and Implementation 17
  4.1 Baseline Algorithm ........................................ 17
  4.2 Improvements ............................................... 18
    4.2.1 Early Proposals .................................... 18
    4.2.2 Overlapped Messages ............................... 19
    4.2.3 Bulk Requests .................................... 20
  4.3 Proposed Algorithm ....................................... 20
    4.3.1 Block structure .................................... 20
    4.3.2 Supported message types ........................... 21
    4.3.3 Algorithm ....................................... 21
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 Software Design ....................................................... 23</td>
</tr>
<tr>
<td>5  Results and Discussion .............................................. 25</td>
</tr>
<tr>
<td>5.1 Experimental Setup .................................................... 25</td>
</tr>
<tr>
<td>5.2 Results ................................................................. 26</td>
</tr>
<tr>
<td>5.2.1 Increased Throughput ............................................... 26</td>
</tr>
<tr>
<td>5.2.2 Reduced Number of Messages ...................................... 27</td>
</tr>
<tr>
<td>5.2.3 Time Difference between Successive Blocks .................... 27</td>
</tr>
<tr>
<td>5.2.4 Impact of Faulty Leaders ........................................... 28</td>
</tr>
<tr>
<td>6  Conclusion .............................................................. 29</td>
</tr>
<tr>
<td>6.1 Future Work ............................................................ 29</td>
</tr>
<tr>
<td>6.1.1 Deploying Different Topologies ................................... 29</td>
</tr>
<tr>
<td>6.1.2 Security Analysis .................................................... 30</td>
</tr>
<tr>
<td>6.1.3 Incentive to be Honest .............................................. 30</td>
</tr>
<tr>
<td>Bibliography ............................................................... 31</td>
</tr>
</tbody>
</table>
List of Figures

2.1 A typical blockchain network ........................................... 6
2.2 A malicious lieutenant attempts to confuse an honest lieutenant [Lamport et al., 1982] .................................................. 8
3.1 Normal case operation of PBFT [Castro et al., 1999] .............. 14
4.1 Comparison of normal PBFT to the improved algorithm with early proposals .............................................................. 19
4.2 Comparison of normal PBFT messages to CollectiveMessages .... 20
4.3 Software design for the proposed algorithm .............................. 23
5.1 Comparison of throughput with varying network sizes .............. 26
5.2 Comparison of time taken between successive block generation (with varying network sizes) ................................. 27
List of Tables

5.1 Benchmarking results ...................................................... 26
5.2 Average number of messages passed from the leader to a network of
  10 replicas ........................................................................ 27
5.3 Number of blocks generated in networks with a single byzantine leader.
  (Each test was run for a minute) .......................................... 28
Introduction

In this chapter, we describe the need for energy-efficient consensus algorithms in the blockchain space. In section 1.2, we also discuss the benefits of using Practical Byzantine Fault Tolerance (PBFT) for blockchain consensus and understand the motivation for improving PBFT. In section 1.3, we summarize the main contributions of this research. Finally, the outline of the thesis is detailed in section 1.4.

1.1 Motivation

Blockchains can be thought of as a distributed ledger that holds immutable data. The ledger cannot be modified since the data (or equivalently blocks) is secured using cryptographic techniques. Each block in the ledger holds details of its relationship with previous blocks in the ledger. Consequently, blockchain technology deals with managing a chain of encrypted blocks that are replicated over multiple locations. This means that blockchains are also decentralized and hence, require a trustless mechanism to agree on the ordering of blocks. Consensus algorithms provide this trustless protocol for a blockchain system to operate. As a result, consensus algorithms are the key to making blockchains a secure decentralized and distributed ledger.

Bitcoin is the first blockchain to be developed and uses Proof of Work (PoW) for consensus [Nakamoto et al., 2008]. Due to its huge success, PoW became very popular in the blockchain space. While PoW has proven to be secure over time, it also has many downsides. The biggest one is the fact that PoW is highly energy intensive and thus, expensive and harmful to the environment [O’Dwyer and Malone, 2014]. These problems eventually triggered the search for alternative consensus algorithms that are as secure as PoW while being cleaner and cheaper to operate.

1.2 Problem Statement

Practical Byzantine Fault Tolerance (PBFT) was introduced in 1999 as a solution for the Byzantine Generals Problem in real-world distributed systems [Castro et al., 1999].
In a blockchain context, it has two major advantages over PoW consensus. Firstly, unlike PoW, PBFT does not require large amounts of energy to produce a block. Also, with PBFT, once a block is confirmed by the network, it does not have to be verified. In other words, block finality is instant, unlike PoW. In PoW, blocks must be verified by other nodes in the network. And they are secured in the blockchain as more blocks are built on top of them. Also, since PBFT is proven to be secure in systems with up to $n/3$ malicious actors, it is secure enough to be used for consensus in blockchains. However, PBFT also has drawbacks, which has restricted the adoption of PBFT in large scale real world systems. PBFT is very network intensive and can only be used with smaller networks.

We address this problem by improving the existing PBFT algorithm to make it less network intensive and more usable for blockchain systems. PBFT is a three-phase voting algorithm. In blockchains, each block is added to a ledger in a chronological order. After a block is added, its data is used to build successive blocks. Similarly, with PBFT, once a block is committed by a majority in the network, the next block can be proposed. We can, however, modify PBFT to propose new blocks at the end of the second phase. This will commence consensus work in parallel for the next block while the previous block is being committed. Furthermore, this improvement should ideally increase the throughput of blocks that are appended to the blockchain and, thus, make the algorithm faster. Building upon this improvement, we can also introduce a new message type that can collect multiple messages for different blocks. This would reduce the number of messages that are required to be passed for the consensus protocol. Also, since the PBFT’s view-change protocol requires messages to have at least $2f + 1$ prepares, we can use Tendermint’s approach for handling Byzantine leaders.

1.3 Contribution

The main contributions of this report are as follows:

- We build a baseline PBFT algorithm (with inspiration from other consensus algorithms in the blockchain space) that can be used for blockchains.

- We modify the baseline algorithm to reduce the amount of communication required and improve the throughput of blocks generated.

- We evaluate the performance gains achieved with the improved algorithm

- Experimental results shows that there was an average of 40% increase in block generation and a 30% decrease in messages passed across the network.
1.4 Report Outline

This thesis has 6 chapters in total. In Chapter 2, we give an overview of the blockchain technology and briefly discuss some of the most popular blockchain consensus algorithms. Next, Chapter 3 describe the Practical Byzantine Fault Tolerance algorithm. We also provide preliminary knowledge for understanding the Tendermint Core consensus algorithm, which is an adaptation of PBFT. In Chapter 4 we introduce the baseline algorithm that combines features of both Practical Byzantine Fault Tolerance algorithm and Tendermint. We also discuss the improvements that can be made to this baseline algorithm and describe the new proposed algorithm that incorporates the improvements. In Chapter 5, we detail the experiment setup and present the results obtained. We also analyse the performance gains achieved. Finally, in Chapter 6, we conclude the thesis findings and discuss possible future work.
Introduction
Chapter 2

Background

This chapter introduces blockchain technology and discusses one of the biggest challenges faced by decentralized and distributed systems, namely; achieving byzantine fault tolerance. We also discuss some of the popular blockchain consensus algorithms.

Section 2.1 focuses on the blockchain technology which will help to understand the need for consensus algorithms in the space.

Section 2.2 introduces a classic problem of distributed systems, the Byzantine Generals Problem, which will form the foundation for understanding Byzantine Fault Tolerance in blockchains.

Section 2.3 introduces some of the most popular consensus algorithms that are used in present blockchains.

2.1 Blockchain

In the most basic form, a blockchain can be viewed as a database storage. Blocks in a blockchain can consist of any kind of data that has some value. The primary reason that a blockchain is named as such, is because it is a chain of blocks that are ordered in a chronological fashion.

A blockchain, however, provides many more features than a simple database, the most important of which is that data within a blockchain cannot be corrupted. This is because data is collected into blocks, which is first linked to previous blocks and then secured using cryptographic techniques. As a result, even the smallest change in any block’s contents would corrupt all following blocks, thus, invalidating the blockchain. Also, blockchains are truly decentralized because they are managed by the users of the system [Crosby et al., 2016]. Therefore, unlike centralized systems such as banks, governments, etc; blockchains do not require users to place their trust on a central authority. Each user is accountable for their actions and managing their assets. Furthermore, blockchains promote transparency because everyone in the sys-
Background

A blockchain is a technology that has a copy of the ledger. Consequently, unlike centralized systems, there isn’t a central server (or database) that can serve as a single point of failure. Figure 2.1 depicts a typical blockchain network where each user manages a copy of the ledger and participates in the network to keep this copy updated.

Figure 2.1: A typical blockchain network

Consensus is a core feature of any blockchain technology [Crosby et al., 2016]. As blockchains are distributed and decentralized systems, they are vulnerable to the biggest problem that affects all distributed systems, that is, collectively arriving at the same decision at any given point. Specifically, users of a blockchain need to agree on the ordering of blocks in a decentralized manner. This is fairly trivial problem to solve if the system has honest participants and uses a reliable network. However, in practical real world systems, this is never the case. In the presence of malicious entities and/or network failures, blockchains must be able to operate smoothly without compromising its security. Consensus algorithms were introduced to the blockchain ecosystem to solve this problem. The presence of such algorithms ensure that users participating in the system do not have to trust anyone but the algorithm that the system is based on.
2.2 Byzantine Fault Tolerance

In the previous section, we briefly discuss the difficulty of arriving at consensus in a distributed system with malicious parties. In this section, we detail this problem, which is also known as the Byzantine Generals’ Problem. Next, we understand how blockchains are vulnerable to this problem. Finally, we define Byzantine Fault Tolerance with reference to the Byzantine Generals’ Problem in the blockchain context.

2.2.1 Byzantine Generals’ Problem

The Byzantine Generals Problem was formulated in 1982 by Leslie Lamport et al. [Lamport et al., 1982]. The problem is as follows:

Assume that a group of generals and their armies are preparing to attack a castle. In order to launch a successful attack the generals must attack at the same time. This implies that they must arrive at a collective decision regarding the attack and they can only communicate using messengers. For this purpose, there exists a commander who will decide to either attack or retreat and all other generals act based on this decision. However, there could be traitors amongst the generals and this traitor could also be the commander. There is also a possibility that messengers could be captured and messages would not be delivered at all. Understandably, the problem now boils down to how generals can arrive at the same decision in the presence of a small number of traitors.

Figure 2.2 shows how a malicious general can disrupt the decision of an honest general. This problem can be solved if all honest participants can rely on an algorithm and this algorithm should ensure that they will agree on the same decision. This algorithm should also guarantee that the presence of a small number of malicious generals will not tamper with the decision making process of honest generals.

Byzantine Generals’ Problem in Blockchains

As discussed in the previous sections, blockchains are distributed systems and therefore, are affected by the Byzantine Generals’ Problem. Since users in a blockchain system do not trust each other and are responsible for managing their own assets, they need to use a trustless mechanism for arriving at consensus to add blocks in the blockchain. Here, each general is a user that is participating in the blockchain. As a result, an algorithm that ensures all honest users agree to add the same block is critical for the system to operate smoothly. This algorithm must also ensure that no honest user adds a different block to its blockchain under the influence of a small number of malicious users. This can only be solved if all honest participants follow
Background

Figure 2.2: A malicious lieutenant attempts to confuse an honest lieutenant [Lamport et al., 1982]

an agreed upon consensus algorithm.

2.2.2 Byzantine Fault Tolerance

In blockchains, a byzantine fault is a user that presents conflicting information to the network. In a real world system, it is difficult for an honest user to determine whether another user is malicious, inactive or honest. If a distributed system is able to arrive at the same decision in the presence of a threshold of byzantine faults, it is said to be Byzantine Fault Tolerant. There are many attacks that can threaten the security of a blockchain system. In the following section, we discuss the double spend attack, which is one of the most crucial attacks that a blockchain is vulnerable to.

Double spend attacks

Byzantine faults are a very serious concern because of the value that each block holds in the blockchain. This value incentivizes malicious actors to purposefully tamper with block consensus and thus make economic gains in the process. An attack that is based on this is the double spend attack.

The double spend attack can be understood with the following example:

Alice creates a transaction "I have 50 apples and will transfer them to Bob in exchange for his 50 bananas" and sends this transaction to half of the users in the blockchain. She then creates another transaction "I have 50 apples and will transfer them to Charlie in exchange for his 50 carrots". Because
the information that Alice has 50 apples is public, all honest nodes will be able to validate the transactions they have received. However, if the system is not Byzantine Fault Tolerant, both transactions will get executed and Alice will end up getting 50 bananas as well as 50 carrots from Bob and Charlie. But since Alice only had 50 apples, either Bob or Charlie will not receive their payment.

The Byzantine Generals Problem has been solved for blockchains by adopting various Byzantine Fault Tolerant consensus algorithms. Some of the most popular consensus algorithms will be discussed in detail in the next section.

## 2.3 Consensus Algorithms

As we have understood from the previous sections, consensus algorithms are crucial for blockchains to operate securely. A consensus algorithm defines a protocol that all participants of the system must adhere to in order to reach a collective non-conflicting decision. In blockchains, this decision will decide the next block that must be added to the blockchain. Furthermore, once this agreement has been finalized, there can be no backtracking from the decision. Thus, a byzantine fault tolerant consensus algorithm will ensure that the blockchain is secure and is immune to attacks.

### 2.3.1 Proof of Work

Proof of Work was originally developed as a mechanism for a prover to prove to a verifier that they have done some work [Dwork and Naor, 1992]. The work is essentially a problem with some degree of computational difficulty and would require a considerable amount of time to solve. However, once the result is obtained, it is expected to be fairly easy to verify its correctness. This concept was then modified to develop HashCash, a mechanism for preventing spam emails [Back, 1997]. The computational problem developed for HashCash was to find a value that when combined with the email contents, produces a hash with a fixed number of leading zeros. This number is essentially called the Proof of Work (PoW) because while it is computationally very difficult to generate, it is trivial to verify.

**PoW for Consensus**

The concept of blockchain was invented by Satoshi Nakamoto for the digital currency called Bitcoin [Nakamoto et al., 2008]. Since Bitcoin used PoW for making the system Byzantine Fault Tolerant, PoW is also known as the first blockchain consensus algorithm [Lab].

In Bitcoin, Proof of Work consensus works by having participants (also known as miners) find a number that when combined with the block contents, creates a hash
code with a predefined number of leading zeros. The difficulty of the problem is adjusted every 2016 blocks. This is done to ensure that a new block is generated every 10 minutes on an average. Also, when nodes receive 2 valid blocks, honest nodes will only pick the longer chain because the work they do on that will be eventually accepted as the true state of the blockchain. By doing this, a malicious miner must have more than 50% of the network’s computational power to generate a chain longer than the current chain. Participants that solves the problem receives bitcoin along with some transaction fees. As a result, miners are incentivised to generate more blocks rather than waste time and energy in trying to attack the system by producing malicious blocks.

Disadvantages of PoW

Proof of Work has been widely criticized to be expensive to operate and energy inefficient. In mid-2018, it was estimated that the Bitcoin network used at least 2.55 gigawatts of electricity, and could potentially increase to 7.67 gigawatts in the future. This makes the energy requirements for operating Bitcoin comparable with energy consumption of countries such as Ireland and Austria [De Vries, 2018]. Moreover, in order to mine bitcoins, miners are expected to purchase dedicated hardware, thus adding to the overall expenses. As a result, Bitcoin’s PoW is considered to be inefficient, expensive and wasteful.

Another drawback of using Proof of Work for consensus is its potential vulnerability to attacks such as the 51% attack. A 51% attack occurs when a group of miners has control over more than 50% of the network’s mining power. Consequently, they could stop transactions from being executed or even revert previous transactions and double spend those coins. Presently, this attack is impossible on Bitcoin due to the availability of a large number of miners. However, this attack can be easily conducted on smaller blockchains with a lesser number of miners.

2.3.2 Proof of Stake

Proof of Stake is a consensus algorithm that was developed to combat the limitations of Bitcoin’s PoW [King and Nadal, 2012]. Proof of Stake (PoS) is a cleaner alternative to achieving consensus in blockchains. In PoS, users that participate in maintaining the blockchain are called validators, and instead of spending energy in producing a block, validators are expected to stake funds in order to be eligible to propose blocks to the blockchain. Thus, a node with 10% of the total funds will be responsible for generating 10% blocks in the blockchain. If a validator is found to be malicious, their bonds are taken away from them. As a result, staking a bond ensures that participants are incentivized to act honestly.
Evidently, PoS does not require the energy that a PoW system needs and thus, is cleaner and cheaper to operate. Furthermore, PoS is not in danger of the 51% attack that PoW is vulnerable to. To perform such an attack on a PoS blockchain, a malicious user must have at least 51% of the total network funds staked. However, in such a scenario, the user will not be incentivized to attack the system as this will impact the value of the currency that the user is a major stakeholder of. As a result, rather than attacking the blockchain, it would be in the user’s best interests to keep the blockchain as secure as possible [King and Nadal, 2012].

Disadvantages of PoS

While PoS overcomes some of the major drawbacks of PoW, it has a number of attacks that it could be potentially vulnerable to. The most popular is the *Nothing at Stake* attack. In this attack, when there is a consensus failure, validators have nothing to lose by supporting multiple blocks.\(^1\) Also, because PoS is a relatively newer consensus algorithm, it has not been widely tested for security implications and is under scrutiny with new vulnerabilities being identified until very recently.

\(^1\)For a detailed explanation of this attack, please refer: https://github.com/ethereum/wiki/wiki/Problems
Background
In this chapter, we discuss Practical Byzantine Fault Tolerance (PBFT) and its applications in the blockchain space. Section 3.1 provides an in-depth explanation of the PBFT algorithm. We also discuss how PBFT handles byzantine leaders with the view-change protocol. Next, Section 3.2 details the Tendermint protocol, which is a variation of the PBFT algorithm that was developed specifically for blockchains.

### 3.1 Practical Byzantine Fault Tolerance

The Practical Byzantine Fault Tolerance (PBFT) algorithm was proposed by Miguel Castro in 1999 [Castro et al., 1999]. PBFT is the first consensus algorithm that was developed to solve the Byzantine Generals’ Problem for real world distributed networks.

In PBFT, all participants progress through views. A single node is selected as the leader for a particular view, while all nodes become replicas (also known as backups). In order to agree on a block, all participants in the network advance through a three-phase message passing consensus protocol.

#### 3.1.1 Algorithm

**Normal Case Operation**

For a network with \( n \) participants, the algorithm proceeds as follows:

1. The client sends a request to all the replicas.

2. The leader for the current view accepts the request and prepares a *PrePrepare* message with the request embedded in it. The leader then broadcasts this *PrePrepare* to all the other replicas in the network.

3. On receiving a *PrePrepare* message, a replica first validates the contents and then, generates a *Prepare* message for that request and broadcasts this to the network.
4. On receiving at least $2f$ Prepares, a Commit is generated and is broadcast to the network.

5. When a replica receives at least $2f + 1$ Commits, it will execute the request and send the reply back to the client.

In this algorithm, $f = (n - 1)/3$ and the three phases are the PrePrepare, Prepare and Commit rounds. Figure 3.1 depicts the normal case operation of PBFT. The PrePrepare and Prepare phases ensure that requests are ordered across the same view, while the Prepare and Commit phases ensure that requests are ordered across multiple views. A network (with up to $f$ malicious nodes) that observes PBFT consensus algorithm is guaranteed to be byzantine fault tolerant.

![Normal case operation of PBFT](castro_et_al_1999)

**Figure 3.1**: Normal case operation of PBFT [Castro et al., 1999]

**View Changes**

To handle faulty replicas, PBFT has a view change protocol. On receiving a request for view $i$, all replicas start a timer which will be stopped once it executes the request. At the event of a timeout, replicas will broadcast a ViewChange message for view $i + 1$ that will consist of the last stable checkpoint (known to that replica) and a set of $2f + 1$ valid checkpoint messages to prove the checkpoint’s correctness. The ViewChange will also contain a set of messages for every request that was prepared at view $i$. When the replica for view $i + 1$ receives $2f$ valid ViewChange messages, it prepares a NewView message and broadcasts it to the network. During a view-change, all other messages (PrePrepare, Prepare and Commit) are ignored by the network.
3.1.2 Advantages and Disadvantages

As evident from the algorithm, PBFT consensus is not computationally intensive. Thus, it clearly consumes much less energy when compared to Proof of Work. Also, once a block is signed and committed by more than $2n/3$ replicas in the network, it cannot be updated. As a result, unlike PoW where at least six block confirmations are required to guarantee finality, in PBFT block finality is instant.

However, PBFT also has a few drawbacks [Chondros et al., 2012]. It is a very network intensive algorithm as it requires $n^2$ messages to be passed around to arrive at consensus for a single block [Liu et al., 2019]. This is a problem especially when it comes to scaling the network to support a larger number of nodes. Another well-known vulnerability of PBFT is its vulnerability to Sybil attacks. A Sybil attack occurs when either a group of nodes or a single user with multiple node identities collude to form a majority in the network and therefore, threaten the security of the system [Sukhwani et al., 2017]. In blockchains, this attack can be prevented by integrating Proof of Stake algorithm along with PBFT.

Since PBFT is a network intensive algorithm, this project targets to reduce the number of messages that are passed around by combining Commit of a block at height $H$ with Prepare of a block at height $H+1$. This will be detailed in the following chapters.

3.2 Tendermint Core

Tendermint Core, developed by Jae Kwon and Ethan Buchman, is a consensus algorithm inspired by PBFT [Buchman et al., 2018]. Each round in Tendermint follows a pattern similar to the three phase communication scheme in the "normal-case" PBFT. As a result, when the system has a majority of honest participants with a reliable network, Tendermint follows exactly from PBFT.

3.2.1 Main Differences from PBFT

Tendermint differs from PBFT in certain aspects. First, unlike PBFT’s View Change protocol, Tendermint has a novel approach to handling byzantine faults. Second, Tendermint rotates the leader after every round which makes it less sensitive to a malicious leader. That is, if an elected leader is Byzantine and maintains strong network connections with other nodes, it can compromise the security of the system by dropping transactions [Buchman, 2016]. Third, unlike PBFT, where a client sends new transactions directly to all nodes in the network, in Tendermint, messages are disseminated to replicas (or validators as known in Tendermint) using a gossip protocol [Cachin and Vukolić, 2017].
Tendermint also uses Proof of Stake to protect the system from Sybil attacks. As a result, each validator will have to lock in some funds to be eligible to participate in the blockchain system. Also, Tendermint is a production-ready software for blockchain consensus while PBFT is more of a prototype.

### 3.2.2 Algorithm

A group of validators form the Tendermint network. As mentioned earlier, a single round comprises of three phases. In each round, a leader is selected in a round-robin fashion.

For a network with \( n \) participants, the Tendermint protocol proceeds as follows:

1. Transactions are sent to validators via a gossip network.

2. The leader (starting at round 0) fetches transactions and prepares a *Propose* block with transactions embedded in it. The *Propose* is then broadcast to validators in the network.

3. When a validator receives a *Propose* for a block at the current height, it first validates the contents and then generates a *Prevote* for the block and broadcasts this to the network.

4. On receiving more than \( 2n/3 \) *Prevotes* for a given block at the current height, a *Precommit* is generated and broadcast to the network.

5. When a replica receives more than \( 2n/3 \) *Precommits* for a block at the current height, it will add the block to its blockchain and increment its current height by one.

When a validator receives more than \( 2n/3 \) of *Prevotes*, it enters a state (termed as a *Polka*) on that block and broadcasts a *Precommit* for the block. On reaching a *Polka*, the validator will lock the block that it *Precommits*. The validator will only unlock when it sees a *Polka* for a new block at a later round.

As mentioned earlier, the biggest difference of Tendermint from PBFT is that Tendermint does not use a separate *View Change* protocol to change proposers. In case of byzantine faults such as offline/malicious proposers, partitioned network, etc; validators can broadcast special *nil* prevotes and precommits. This is implemented in Tendermint by using special timeouts for each phase, namely, *timeoutPropose*, *timeoutPrevote*, and *timeoutPrecommit*. When validators time out at the *Propose* phase, they will issue a *Prevote(nil)*. Similarly, if validators time out waiting for a *Polka* on a valid block, they will issue a *Precommit(nil)*. Finally, if validators time out waiting for \( 2n/3 \) *Precommits* for a valid block, they will move onto the next round.
Design and Implementation

In this chapter, we present the design and implementation of the algorithm proposed in this report. Section 4.1 introduces the baseline algorithm that is primarily adapted from PBFT with some concepts of Tendermint. Section 4.2 proposes improvements that can be made to the baseline algorithm. The proposed improvements aim to mitigate the issues surrounding PBFT, as discussed in previous chapters. Section 4.3 presents the proposed algorithm with the discussed improvements. Finally, section 4.4 describes the software design used to implement the proposed algorithm.

4.1 Baseline Algorithm

In order to understand the performance improvements that can be gained from the proposed algorithm, we will first develop a baseline that can be improved upon and be used for benchmarking purposes. This research commenced with the development of a naive PBFT implementation for blockchains. The implementation was later altered to include certain features of the Tendermint consensus algorithm.

As with the original PBFT algorithm, we will use a three-phase consensus algorithm. A height field (denoted by $h$) will be used to keep track of the current height of the blockchain. While processing blocks for a given height ($h$), all messages that arrive for heights lower than $h$ will be ignored and all messages that arrive for heights greater than $h$ will be buffered. The buffered messages will be retrieved by the replica when it starts processing blocks of that height.

Also, instead of the View Change approach, the baseline model was modified to use timeouts. This is an improvement inspired by Tendermint. By introducing timeouts, the algorithm will be able to handle faulty leaders without using an expensive View Change protocol. For example, after committing on a block, if a replica doesn’t receive the next Propose within a predefined amount of time, it will issue a Prepare<nil> and send that to all other replicas. This Prepare will also include the new view number (which would be $v + 1$ where $v$ is the current view). On receiving more than $2n/3$ Prepare<nil>, a replica will publish it’s Commit<nil> for $v + 1$. Once the leader for
\( v + 1 \) receives more than \( 2n/3 \) Commit<\text{nil}>, it will issue a PrePrepare with a valid block for \( v + 1 \).

### 4.2 Improvements

In this section, we propose some improvements to the baseline algorithm. Since the baseline algorithm is adapted from PBFT, it is vulnerable to all the problems discussed in section 3.1. The improvements discussed in this section will primarily attempt to improve the throughput of committed blocks in the baseline algorithm. The proposed algorithm also attempts to reduce the number of messages needed for a block to be committed by the network.

#### 4.2.1 Early Proposals

In PBFT, once a replica receives more than \( 2n/3 \) Prepares and more than \( 2n/3 \) Commits for the same request, the request is said to be confirmed and thus, can be executed. The last two phases of the algorithm (that is, the prepare and commit phases) are present to guarantee the ordering of transactions. However, after receiving more than \( 2n/3 \) Prepares, we can expect the block to be eventually committed by more than \( 2n/3 \) of honest replicas. This idea can be further leveraged to send proposals at an earlier stage. That is, when the leader gets more than \( 2n/3 \) Prepares for a block at height \( h \), it can prepare and broadcast the next PrePrepare block for \( h + 1 \) along with the Commit for \( h \). In this manner, we can increase the throughput of blocks that are finalized. This means that as long as the system is mostly honest, every block after the first block will get finalized at an earlier time than when compared to the normal case operation of PBFT.

As discussed earlier, in the presence of byzantine faults, replicas that do not receive enough Prepares for a valid block will issue a Prepare<\text{nil}>. As a result, when a replica receives more than \( 2n/3 \) of Prepare<\text{nil}> at the current view \( v \), and if the replica is the leader for the new view \((v + 1)\), it will send a PrePrepare for \( v + 1 \). However, as all honest participants will only act on the PrePrepare for \( v + 1 \) after getting more than \( 2n/3 \) of Commits<\text{nil}> for \( v \), this will not affect the ordering of blocks. As a result, we can eliminate the need for expensive View Change messages to handle byzantine faults.

Figure 4.1 shows a comparison between the normal case PBFT and a PBFT with the early proposals improvement. Evidently, with early proposals, replicas can begin processing blocks for heights (greater than the current height), thus, increasing throughput.
### §4.2 Improvements

#### (a) Normal case PBFT

![Normal case PBFT](image)

#### (b) Early proposals

![Early proposals](image)

**Figure 4.1**: Comparison of normal PBFT to the improved algorithm with early proposals

#### 4.2.2 Overlapped Messages

As discussed in the previous chapter, one of the main drawbacks of using PBFT algorithm is that the algorithm is highly network intensive. In order to add a single block to the blockchain, around \( n^2 \) messages will have to be passed across replicas. The baseline algorithm can be modified to alleviate this problem by overlapping messages for the next height along with messages for the current height. A new message type, `CollectiveMessage`, was introduced for this purpose. This message will contain a list of messages supported by the algorithm. For instance, a `Prepare` for height \( h + 1 \) can be piggybacked along with the `Commit` for height \( h \). The `CollectiveMessage` for this will look like:
20

**Design and Implementation**

*CollectiveMessage[Commit(h),Prepare(h + 1)]*

This improvement should ideally reduce the number of messages that are sent across the network. This works particularly well in the algorithm because of the early proposals improvement. In order to make the best use of this new message type, replicas will wait for a very short period of time, before broadcasting any CollectiveMessages to the network. This is to allow for any new PrePrepares, Prepares, or Commits that are ready, to be added to the list of messages in the CollectiveMessage.

![Diagram of message flow](image)

(a) Messages in PBFT (without piggy-backing)

(b) CollectiveMessages that piggy-back blocks of future heights

**Figure 4.2:** Comparison of normal PBFT messages to CollectiveMessages

Figure 4.2 depicts how a CollectiveMessage can be used to overlap messages from multiple rounds. As mentioned earlier, this improvement leverages the Early Proposal improvement to send PrePrepares for blocks from the future heights.

### 4.2.3 Bulk Requests

In traditional PBFT, each client request undergoes the entire three-phase consensus protocol. However, with blockchains, multiple transactions can be allocated within a single block. Thus, instead of using a single client request in a block, transactions can be batched for faster consensus. Thus, execution of transactions can begin faster, when compared to PBFT.

### 4.3 Proposed Algorithm

#### 4.3.1 Block structure

A typical block in a blockchain consists of transaction data and a reference to the previous block in the ledger. This reference is usually a hash of the previous block’s contents.
4.3 Proposed Algorithm

Our algorithm will use this standard block format along with a few additional fields. These fields are the view (to identify which blockchain participant was the proposer) and a nonce which is the Unix time in seconds. We will also include the height at which the block was added. Along with this, there will also be a signature of the proposer that might be used for verification purposes.

4.3.2 Supported message types

- A PrePrepare block consists of a Block (discussed in the previous section) along with the signature of the proposer.

- A Prepare block consists of a PrePrepare along with the signature of the replica that issued the Prepare.

- A Commit block consists of a PrePrepare along with at least 2n/3 Prepare signatures. This message will also have the signature of the replica that issued the Commit.

- And most importantly, a CollectiveMessage consists of a list of one or more of the above messages and it will be used for all external communication between replicas.

All blocks stored in the blockchain will include at least 2n/3 of both Prepare and Commit signatures.

4.3.3 Algorithm

The proposed algorithm proceeds in views and in each view, v:

1. The leader prepares a block and broadcasts a PrePrepare message to all replicas.

   \[
   \text{CollectiveMessage[PrePrepare}\langle v,h,block_h \rangle] \]

   where \( h \) indicates the current height, and \( block_h \) represents the block at that height.

2. On receiving the PrePrepare message, replicas validate the message and broadcasts a Prepare message.

   \[
   \text{CollectiveMessage[Prepare}\langle v,h,block_h \rangle] \]
3. On receiving more than $2n/3$ Prepares:
   - Each replica generates and broadcasts a Commit message.

   $\text{CollectiveMessage}[\text{Commit}<v,h,block_h>]$

   - The leader generates the Commit message along a new PrePrepare for height $h+1$ and broadcasts both messages.

   $\text{CollectiveMessage}[\text{Commit}<v,h,block_h>, \text{PrePrepare}<v,h+1,block_{h+1}>]$

4. On receiving more than $2n/3$ Commits, every replica adds block $block_h$ to the blockchain. At this stage, the replicas would have also received a PrePrepare for $h+1$ and will broadcast the corresponding Prepare message.

**PrePrepare**

In the first phase of the algorithm, the leader for the current view retrieves pending transactions and generates a well structured Block. This Block will include the view, $v$, height at which the block is created, $h$, and the transactions. This block is then signed by the leader and the signed block is used to create a PrePrepare message. The leader finally broadcasts this PrePrepare message, which will be wrapped inside a CollectiveMessage. Since this is the first round of the consensus protocol, the CollectiveMessage will only consist of the PrePrepare message and can be broadcast immediately.

**Prepare**

In the second phase of the algorithm, replicas that receive PrePrepare from the leader validates the contents of the PrePrepare. After the message is verified, replicas generate a Prepare message that consists of the block signed by the leader for the current view, and signs this message before broadcasting it to the network in a CollectiveMessage.

In the event that a replica has not received a PrePrepare within the timeout period, or if it receives invalid PrePrepares, the replica will send a signed Prepare for a nil block.

Once more than $2n/3$ Prepares have been received by any replica (which is not the leader), it verifies the contents of the Prepare. Here, a valid Prepare is a Prepare with a valid signature. The Prepare message can either have a nil block, or if the block exists, the block must be valid too. After verification of the Prepares, the replica generates a Commit with the block (or nil) and the $2n/3$ Prepare signatures that it has seen. This Commit is then signed by the replica before broadcasting to the network.

The leader of the current view follows the same process as explained, with an additional task. Along with the Commit for the height $h$, it also generates a PrePrepare for $h+1$ and populates both these messages in the CollectiveMessage before broadcasting it to the network.
Commit

When a replica receives more than $2n/3$ Commits for $h$, it finalizes the block and adds it to the blockchain. At this stage, replicas would have ideally received a PrePrepare for $h+1$. As a result, they would have also broadcasted the Prepare message for $h+1$.

§4.4 Software Design

The primary component of the software design is a consensus engine that has subcomponents that handles different stages of the algorithm. The consensus engine receives CollectiveMessages from the network. It then processes each message in the list of messages and forwards them to the appropriate engine for further processing. All messages that have to be broadcasted to the network are signed using ECDSA public key cryptography.

As mentioned earlier, the consensus engine consists of two engines that are responsible for different phases of consensus algorithm; the Preparer and Committer engines. The Preparer accepts all PrePrepares, sends back relevant signed Prepares to the consensus engine. It is also responsible for storing all incoming Prepares. When more than $2n/3$ Prepares has been received for a given height, the Preparer generates a signed Commit message, and sends it back to the consensus engine. The Committer engine is similar to the Preparer engine with the only difference that it accepts Commits and sends back a final committed block when more than $2n/3$ Commits has been seen for a given height. Figure 4.3 depicts the software architecture used.

![Figure 4.3: Software design for the proposed algorithm](image)

This software design plays an important role in implementing the optimizations of the proposed algorithm. The first optimization of generating a proposal on seeing $2n/3$ valid Prepares is implemented by modifying the interaction between the consensus engine and the Preparer. When the consensus engine receives a signed Commit from
the Preparer engine, it generates the next PrePrepare message (if it is the leader for the current view). The second optimization of overlapping messages for different heights is implemented by using CollectiveMessages to combine the Propose at height $h + 1$ and the Commit at height $h$. 
Chapter 5

Results and Discussion

In this chapter, we first provide an overview of the experiments conducted and the setup. We also report the results of the experiments and analyse how each improvement impacts the performance of the baseline algorithm.

5.1 Experimental Setup

In this section, we will go through the experimental setup developed for this project. As discussed in the previous chapter, we had started out by building a baseline model which we will use for the comparison tests and further analysis. Both the baseline and improved algorithms were built using Golang (version 1.12).

For the experiments, we simulate a complete network. We know that messages have to be forwarded to and received from via a network. For this purpose, the consensus engine had defined two interfaces, namely, the Receiver, for receiving messages from the network and the Broadcaster, for broadcasting messages to the network. These interfaces can be implemented by any communication channel and an http client/server implementation was also developed. For the experiments, we ran local instances of replicas within a single machine. For this setup, we implemented a mock client and server using golang channels and goroutines (i.e, threads). The mock server implements the Receiver interface by accepting all messages that are written into its input channel and forwarding it to the consensus engine. The server was also responsible for sending Ticks to the consensus engine at regular intervals. The mock client, on the other hand, broadcasts all outgoing messages from the consensus engine to all other replicas in the system. This is implemented by passing all channels defined by each replica’s server to the client. Thus, when an outgoing message is received by a client, it writes that message to all the channels it has access to. As discussed earlier, we know that only messages of the type CollectiveMessage is propagated in the network. This means that the server defined channel restricts messages to be of type CollectiveMessage.
5.2 Results

Both consensus models were run with networks of varying sizes for two minutes each. Each test was executed ten times before computing the average results. The results obtained are presented in Table 5.1. The average number of blocks generated by each algorithm and the average time taken to produce the next successive block is presented in this table.

<table>
<thead>
<tr>
<th>Replicas</th>
<th>Baseline Model</th>
<th>Improved Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of Blocks</td>
<td>Time b/w each block</td>
</tr>
<tr>
<td>10</td>
<td>576.2</td>
<td>0.22</td>
</tr>
<tr>
<td>50</td>
<td>100.1</td>
<td>0.81</td>
</tr>
<tr>
<td>75</td>
<td>69.3</td>
<td>1.70</td>
</tr>
<tr>
<td>100</td>
<td>48.3</td>
<td>2.28</td>
</tr>
<tr>
<td>150</td>
<td>31</td>
<td>3.58</td>
</tr>
<tr>
<td>200</td>
<td>22.7</td>
<td>4.76</td>
</tr>
</tbody>
</table>

Table 5.1: Benchmarking results

5.2.1 Increased Throughput

As seen in Figure 5.1, there is an increased throughput for the improved consensus algorithm. Based on the results obtained, there was an increase in throughput of at least 30% and the average increase in throughput was calculated to be 40%.

Figure 5.1: Comparison of throughput with varying network sizes
5.2.2 Reduced Number of Messages

Table 5.3 presents the number of messages that a leader (proposer) broadcasted in a network of 10 replicas. On an average, there was around 30% reduction in the number of messages sent by a leader to the network. Evidently, the addition of a CollectiveMessage type to the system helped in reducing the number of messages in the network.

<table>
<thead>
<tr>
<th>Number of Blocks</th>
<th>Baseline Model</th>
<th>Improved Model</th>
<th>Percentage Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>326</td>
<td>223</td>
<td>-31.5</td>
</tr>
<tr>
<td>20</td>
<td>616</td>
<td>440</td>
<td>-28.5</td>
</tr>
<tr>
<td>50</td>
<td>1503</td>
<td>1013</td>
<td>-32.6</td>
</tr>
<tr>
<td>100</td>
<td>2936</td>
<td>2000</td>
<td>-31.8</td>
</tr>
</tbody>
</table>

Table 5.2: Average number of messages passed from the leader to a network of 10 replicas

5.2.3 Time Difference between Successive Blocks

Figure 5.2 compares the time taken to generate successive blocks for both algorithms. It can also be observed from these results that the time between a block and its successive block was less for the improved algorithm when compared to the baseline model.
5.2.4 Impact of Faulty Leaders

Table 5.3 compares the number of blocks that were generated with a faulty/offline leader. With a faulty leader in the system, the proposed algorithm was able to continue producing more blocks than the baseline algorithm. This is because both algorithms follow the same protocol for byzantine leaders. As a result, once an honest replica takes over as the leader, there was no significant impact on the performance of the improved algorithm, when compared to the baseline algorithm.

<table>
<thead>
<tr>
<th>Number of replicas</th>
<th>Baseline Model</th>
<th>Improved Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td>100</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>150</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>200</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 5.3: Number of blocks generated in networks with a single byzantine leader. (Each test was run for a minute)
Conclusion

In this report, we developed a consensus algorithm that is inspired by PBFT and Tendermint. Then, we attempted to improve the developed algorithm for blockchain systems. We found that piggybacking messages for blocks of later heights along with the current block’s messages significantly reduced the number of messages that were sent across the network. We also noticed that sending new proposals at an earlier stage of the algorithm increased the block generation rate.

6.1 Future Work

In this section, we will present avenues that can be pursued for future work.

6.1.1 Deploying Different Topologies

Future work can analyse the impact that various topologies can have in improving the performance of the system. For instance, during the research, I had thought of implementing the proposed algorithm on a network with a star topology. With this change, all replicas are expected to be actively connected to the leader. Thus, all messages will go through the leader. This means that the leader will send a PrePrepare to all replicas and once the leader receives $2n/3$ Prepares, it will send a Commit to the replicas and once $2n/3$ Commits are received by the leader, it will send a message aggregating this information to all other replicas. With this approach, the CollectiveMessage could be leveraged to send a PrePrepare at height $h + 2$, a Prepare at height $h + 1$, and a Commit at height $h$; all at the same time. This would also drastically reduce the number of messages that have to be passed among replicas. However, this change immediately assumes that the network is synchronous, and hence, this idea was not explored further. Also, owing to time restrictions, the benefits of using other topologies was not analysed.
6.1.2 Security Analysis

Because the three phase protocol of the original PBFT algorithm is used without significant modification, the security proofs of PBFT’s normal case remains unchanged. However, the proposed algorithm uses a different approach to handling Byzantine faults. This approach is inspired by Tendermint Core. Since Tendermint does not have peer-reviewed security proofs for their consensus algorithm yet, the proposed algorithm cannot be considered absolutely secure. Future work will involve building security proofs for the proposed algorithm and conducting detailed analysis for potential attacks on the system.

6.1.3 Incentive to be Honest

Following from the Proof of Stake algorithm, as an additional security constraint, all replicas can be asked to lock in a bond amount which will be taken from them if they are observed to act maliciously. In this way, the leader selection algorithm can be modified to prefer nodes with more funds at stake, similar to the PoS algorithm. This incentivizes participants to remain honest and propose blocks non-maliciously. As a result, this improvement should provide extra security to the algorithm. However, employing this requires a detailed security analysis, as PoS is still undergoing research in the blockchain space for potential vulnerabilities.
Bibliography


(cited on pages 1 and 9)

(cited on page 1)

INDEPENDENT STUDY CONTRACT
PROJECTS

Note: Enrolment is subject to approval by the course convenor

SECTION A (Students and Supervisors)

UniID: u6479594

SURNAME: FIRST NAMES: Divya Mary

PROJECT SUPERVISOR (may be external): Prof. Uwe Zimmer, Loong Wang

FORMAL SUPERVISOR (if different, must be an RSSCS academic): Prof. Uwe Zimmer

COURSE CODE, TITLE AND UNITS: COMP8755, Individual Computing Project, 12 units

COMMENCING SEMESTER S1 S2 YEAR: 2018 Two-semester project (12u courses only): 

PROJECT TITLE:
Improved PBFT for faster consensus in blockchain systems

LEARNING OBJECTIVES:
* Gain in-depth knowledge of consensus protocols in distributed and decentralized systems.
* Deepen my understanding of Practical Byzantine Fault Tolerance.
* Deepen my understanding of developing blockchain technology.
* Deepen my understanding in applied cryptography.
* Improve my project related skills such as written and oral presentation

PROJECT DESCRIPTION:

Following the Practical Byzantine Fault Tolerance (PBFT) algorithm, nodes in a decentralized system can arrive at consensus with three rounds of message-passing. This algorithm can be further optimized to two rounds of communication with the introduction of a new message type. Building a blockchain that uses this variation of the PBFT algorithm would significantly reduce the amount of time that blockchains take to finalize a block of transactions. Moreover, the throughput of the number of blocks
generated would also be expected to increase. This project will study the feasibility of building such a blockchain and also analyse the security risks and other issues that could arise out of it.

ASSESSMENT (as per the project course’s rules web page, with any differences noted below).

<table>
<thead>
<tr>
<th>Assessed project components:</th>
<th>% of mark</th>
<th>Due date</th>
<th>Evaluated by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report: style: _______________________________ (e.g. research report, software description...)</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artefact: kind: software (e.g. software, user interface, robot...)</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation :</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MEETING DATES (IF KNOWN):

Weekly

STUDENT DECLARATION: I agree to fulfil the above defined contract:

................................................................. .........................
Signature Date

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project. I nominate the following examiner, and have obtained their consent to review the report (via signature below or attached email)

Research School of Computer Science  Form updated Jun 2018
Examiner:
Name: Josh Milthorpe
Signature …………………
(Nominated examiners may be subject to change on request by the supervisor or course convenor)

REQUIRED DEPARTMENT RESOURCES:

SECTION C (Course convenor approval)

Signature …………………
Date