8-1-2021

Empirical Performance Evaluation of Consensus Algorithms in Permissioned Blockchain Platforms

Shiv Sondhi
University of Windsor

Empirical Performance Evaluation Of Consensus Algorithms In Permissioned Blockchain Platforms

By

Shiv Sondhi

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science
at the University of Windsor

Windsor, Ontario, Canada

2021

Empirical Performance Evaluation Of Consensus Algorithms In Permissioned Blockchain Platforms

by

Shiv Sondhi

APPROVED BY:

W. Anderson
Department of Political Science

B. Boufama
School of Computer Science

S. Sherif, Advisor
School of Computer Science

July 14, 2021

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

Over the past decade or so, blockchain and distributed ledger technology (DLT) have steadily made their way into the mainstream media. As a result, new blockchain platforms and protocols are emerging rapidly. However, the performance of the resultant systems, and their resilience in hostile network environments is as yet not clearly understood. This thesis proposes a methodology to compare these platforms (specifically permissioned platforms) - and analyze the role of consensus protocols in determining system performance. It studies system performance in the face of network faults and varying loads, and also provides a qualitative analysis of each shortlisted platform.

The four platforms - Ethereum, Hyperledger Fabric, Hyperledger Sawtooth, and Cosmos-SDK - are shortlisted on the basis of the consensus protocols they offer, i.e. Clique, Raft, PBFT, and Tendermint respectively. The following chapters discuss our selection criteria, the performance metrics used for comparison, and the steps followed to build a blockchain application on each platform. Considering the prominence of modelling techniques in the existing literature, we build stochastic models for each shortlisted protocol, and measure the same performance metrics as in our applications. Ultimately, this research aims to determine what factors affect the performance of blockchain systems, and what is the best way to measure their performance characteristics - by building applications or by building stochastic models?

The experiments show that both methods of performance measurement have their pros and cons. They also highlight the importance of platform architecture in the determination of system performance. Selecting consensus protocols and blockchain platforms are critical decisions for any blockchain system. However, different choices shine in different settings. To recognise the best choice for a given use-case, it is crucial to first compare the protocols - and this thesis does that on the basis of performance.

AKNOWLEDGEMENTS

Here, I would like to thank and acknowledge my supervisor, Dr. Sherif Saad for guiding me throughout my master's program and directing me towards this topic of research. I would also like to thank the other members of my committee - Dr. William Anderson who offered valuable feedback during my time working at the Cross-Border Institute; and Dr. Boubakeur Boufama, for his valued opinion and his advice on improving the scope of my research.

I would like to thank my fellow master's student, Kevin Shi, for helping me explore this topic and providing invaluable insights on a regular basis.

I would like to thank Dr. Kobti and the computer science department at the University of Windsor, for their constant support. And finally, I would like to thank my family and my peers for their faith and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

SDK	Software Development Kit
PBFT	Practical Byzantine Fault Tolerance
DLT	Distributed Ledger Technology
DLS	Dwork Lynch Stockmeyer (protocol)
PoW	Proof-of-Work
PoS	Proof-of-Stake
BFT	Byzantine Fault Tolerant
PoA	Proof-of-Authority
PoET	Proof-of-Elapsed Time
CTFG	Casper The Friendly GHOST
GHOST	Greedy Heaviest-Observed Sub-Tree
PER	Packet Error Rate
SDL	Specification and Description Language
DAG	Directed Acyclic Graph
TP	Throughput
L	Latency
SR	Success Rate
σ	Standard Deviation of Local Chains
SHA-256	Secure Hash Algorithm-256
DTCC	Depository Trust and Clearing Corporation
DAO	Decentralised Autonomous Organization

SMR	State Machine Replication
FLP	Fischer-Lynn-Patterson (Impossibility)
CAP	Consistency Availability Partition tolerance (Theorem)
CFT	Crash Fault Tolerant
DoS	Denial of Service
DPoS-BFT	Delegated Proof-of-Stake Byzantine Fault Tolerant
SLA	Service Level Agreement
SQA	Software Quality Assurance
QC	Quality Control
P2P	Peer-to-Peer
EVM	Ethereum Virtual Machine
DApps	Decentralized Applications
AWS	Amazon Web Services
geth	Go Ethereum
JSON	JavaScript Object Notation
RPC	Remote Procedure Calls
API	Application Programming Interface
ABCI	Application Blockchain Interface
CTMC	Continuous Time (discrete state space) Markov Chains
DTMC	Discrete Time Markov Chains
MDP	Markov Decision Process
LTS	Labelled Transition Systems
SHA	Stochastic Hybrid Automata

REST	Representational State Transfer

CHAPTER 1

Introduction

A blockchain is a distributed ledger used to store data. There is no real restriction on the type of data blockchains can store - although some data characteristics (like low memory requirements) may be preferred. They have been used to store health records (health applications), ownership records (property management) and even executable code in the form of smart-contracts. All blockchain functions, like adding, updating and deleting records, are governed by consensus protocols. These protocols define the criteria through which changes to the blockchain state are either accepted or rejected by the network.

The main objective of this research is to compare the performance of a few selected permissioned blockchain platforms and study the role of consensus protocols in performance determination. The experiments compare performance results from blockchain deployments as well stochastic models and evaluate the usefulness of both techniques. Modelling techniques are useful due to their adaptability and time savings, while blockchain deployments give a better understanding of real-world system performance. Finally, we aim to analyze the correlation between stochastic elements in the blockchain network, and performance of the blockchain applications. Here, stochastic elements include input load, packet errors, node failures, transmission delays, varying capabilities of nodes, etc.

These objectives help determine a few things. First, they help us learn what factors affect the performance of a blockchain system and the importance of consensus protocols in determining system performance. They also help verify the effectiveness of stochastic modelling techniques with respect to performance measurement. Finally,

they give some insight into how different protocols handle stochastic network elements.

Consensus protocols are a crucial component of blockchain systems - however, there are too many to choose from, and no clear selection guidelines exist. Moreover, appreciating the differences between the various platforms and their protocols requires some technical knowledge. Even after a platform or consensus protocol has been selected, it is important to know how this decision affects the entire system in terms of system performance and security. Hence, comparing the performance delivered by different blockchain platforms, and studying the effect of stochastic elements on performance are both important tasks. Given that decision-makers in industries like finance, healthcare and government come from non-technical backgrounds, it is difficult for them to make informed decisions while choosing a blockchain platform or consensus protocol. This research lays down a methodology to help experts make these critical decisions.

In this research, we build permissioned blockchain applications using four blockchain platforms - Ethereum, Hyperledger Fabric, Hyperledger Sawtooth, and Cosmos SDK. Performance evaluation is used to estimate the loads that each platform can handle, the speeds it can serve, and its ability to run as expected even in the face of failures. These are important considerations while selecting a blockchain platform for any use case. In addition, we evaluate the performance of stochastic models of the consensus protocols that these platforms use i.e. Clique, Raft, PBFT, Tendermint respectively. These probabilistic models use automata theory to run simulations and measure the same performance characteristics that were measured in the blockchain applications.

Based on the above methodology, this thesis provides a comparative analysis of 4 unique, permissioned DLT platforms, and studies the role of consensus protocols and stochastic network elements on their performance. Chapter 2 discusses the existing literature in this domain, Chapter 3 examines our choice of protocols and performance metrics, Chapter 4 describes our methodology and performance evaluation experiments, and Chapter 5 presents our experiment results. Finally, Chapter 6 suggests possible future work and concludes the thesis.

CHAPTER 2

Related Work

This chapter provides a background for the major topics covered in this research - blockchain consensus protocols, performance measurement, and stochastic modelling. The first section discusses the history of consensus protocols and major breakthroughs in this domain. It also talks more specifically about the protocols used in this research. The second section discusses techniques used to study consensus protocols in a blockchain context. The literature includes topics from modelling techniques for blockchain systems, to performance comparison of consensus protocols. Finally, the third section gives a brief overview of stochastic modelling and its use cases. It also touches on how and why researchers have used stochastic modelling to study blockchain systems. Key takeaways from the literature review are provided at the end, and our research methodology is compared to the methodology used by other work in the literature.

2.1 Consensus Protocols

In a distributed system, several machines work together towards a common set of goals. For this, they must work cooperatively and may use some shared resources. However, these machines must be able to agree on the current state of the system - this is the problem solved by consensus protocols. The most straightforward (and early) solution was to vote on the system state by passing messages between machines. The state that received a majority of the votes was selected as the system's current state. The initial protocols required each pair of machines to communicate with eachother

before coming to a final decision resulting in a lower bound of $O(n^2)$ messages being passed before consensus could be reached. For smaller networks, with a reasonable number of nodes, this method worked fine. But as the systems grew larger, the protocol could not scale.

The last two decades of the 20th century saw research into several topics like the Byzantine general's problem [1] and finding consensus in partially synchronous systems [2]. It also saw the creation of two popular consensus protocols - Paxos and Practical Byzantine Fault Tolerance (PBFT) [3]. The Byzantine general's problem outlined the issue of reaching consensus in large distributed systems that contain a number of malfunctioning or byzantine components. As opposed to crashing components, byzantine components may send contradictory information to different parts of the system - which naturally impedes the process of consensus. For instance, a byzantine node may tell one peer that it agrees to the proposed state change, and tell another peer that it does not agree to the change. The motive is for all functioning components to agree on the final decision, but make the decision individually. The paper proves (mathematically) that as long as more than two-thirds of the components are not byzantine, this objective is always met. This means that even a single byzantine node can compromise a network of three nodes.

Paxos was created in the 1990's and is a crash fault tolerant consensus protocol. It was quicker, more scalable and generally better than other protocols available at the time. However, it was hard to understand, and many implementation details were left open to interpretation. This made the protocol hard to implement and the field of distributed systems extremely hard to navigate [4]. Using concepts introduced in [2], the DLS protocol was created. This protocol worked in partially synchronous systems and was also byzantine fault tolerant - a major improvement over Paxos. However, due to security vulnerabilities it was never widely implemented. PBFT, created in 1999, was the first practical implementation of a byzantine fault tolerant consensus protocol. It worked well in asynchronous and partially synchronous systems, and was considerably quicker than existing solutions. It is still used in applications even today.

More recent advances in consensus protocols occurred after the introduction of the

Bitcoin white paper in 2008 [5]. Since then, as the number of consensus protocols has continuously risen, a number of protocol families have appeared. Families are made up of protocols that are similar to eachother. Each protocol family is usually inspired by a single protocol, for instance, the proof-of-work family was inspired by the proof-of-work protocol (now called Bitcoin's proof-of-work) and the proof-of-stake family was inspired by the proof-of-stake protocol. All protocols that belong to a given family work similarly, however, across families several differences emerge. Some of the popular consensus families are listed below with examples [6] and the relevant ones are revisited in Chapter 3.

- **Proof-of-work (PoW-based)** - Bitcoin's PoW, Ethash (Ethereum's PoW).
- **Proof-of-stake (PoS-based)** - Casper / CTFG (Ethereum 2), Delegated PoS.
- **Byzantine fault tolerant (BFT-based)** - PBFT, Ouroboros (Cardano's BFT protocol).
- **Paxos-based** - Paxos, Raft.
- **Proof-of-authority (PoA-based)** - Clique (Ethereum's Rinkeby and Görli testnets), Authority Round (Ethereum's Kovan testnet).

Four papers that were most relevant to this research are outlined below. The papers that introduce protocols are organized in a similar fashion and discuss the algorithm and architecture set-up, provide formal proofs, discuss special cases wherever applicable, and suggest optimizations.

- Castro et al. introduced the PBFT consensus protocol for distributed systems in [3]. The authors assume an asynchronous system with network delays and errors, and the presence of byzantine nodes on the network. The paper describes the algorithm, proves its correctness and safety, and provides optimizations as well as performance evaluations. PBFT used in blockchain applications has only subtle differences from PBFT used in distributed systems.

- Buchman et al. introduce the Tendermint consensus protocol for distributed ledger systems in [7]. The authors provide the algorithm along with proofs of correctness and implementation details. Tendermint is a variant of PBFT, with an improvement in the termination condition (when a new block is added and the leader must change). The paper highlights some differences between Tendermint and other PBFT-based consensus protocols. Tendermint is used widely in decentralised applications.

- In [8], Ongaro et al. introduce the Raft consensus protocol for managing replicated logs and distributed ledgers. The paper describes the different phases of the Raft consensus algorithm, which was developed as an improvement over the Paxos protocol. Raft is more understandable and easier to implement than Paxos. The authors discuss optimizations and handling of events like cluster membership. They also provide a proof of correctness and safety; and touch on performance considerations. Raft consensus is used largely in distributed systems and DLT applications.

- In [9], Sadek et al. provide a detailed survey of consensus mechanisms and their types. The paper provides a taxonomy of consensus protocols including structural, block and reward, security and performance properties. The authors also provide a detailed analysis of two famous incentivised protocols - proof-of-work and proof-of-stake. An incentivised protocol is one where block creators have an incentive for block creation - usually in the form of some reward. They then discuss popular platforms for non-incentivised protocols and finally provide a decision tree to select a protocol based on its properties (incentives, scalability, security and energy consumption). The discussions in this paper are observed more closely in Chapter 3.

2.2 Studying Performance Of Consensus Protocols

Since the nodes of a distributed or blockchain network are working towards a common goal, finding agreement amongst them is a fundamental function of the system. Without a way for blockchain nodes to reach a common conclusion, decentralization of the system would be impossible. It is only because the network can make a consensual decision that blockchain systems can run without a central authority.

Finding agreement is easy when the number of nodes is small, but gets harder as the network size increases. In addition, some nodes may be faulty or byzantine - which makes reaching consensus even harder. These seemingly random behaviours affect the performance and guarantees provided by the blockchain application. Guarantees include consistency, availability, decentralization and partition tolerance. Each guarantee is a continuum rather than a binary value, and it is not necessary that a given system offers all four. For instance, it is very likely that a system which is highly available is not always consistent. This is discussed further in Chapter 3. This section discusses previous works that investigate, compare, and measure the performance of consensus protocols used in blockchain applications.

- In [10], Ilja et al. attempt to compare the BFT-based consensus mechanism in Bitfury's Exonum blockchain framework, to Bitcoin's PoW. They use a toolset called Modest, to model the protocol as distributed stochastic hybrid automata. In the model, packet error rate (PER) is used as the stochastic component, and the minimum and maximum times to commit a block to the blockchain are recorded. The authors built two models with 4 and 7 nodes each. Commit times were plotted against a varying PER, to examine how the two were correlated. Finally, the authors introduce a malicious node in the 4-node model to check the two-third majority voting principle of [1]. The results indicate that time to commit increases almost exponentially as PER increases. On the other hand, in the Bitcoin protocol there is not much effect on commit time - instead a

higher PER, leads to forks in the blockchain which in turn leads to higher read latency and throughput. A fork occurs in a blockchain when two or more nodes create a valid block almost simultaneously. The authors conclude by noting that comparing different consensus protocols is not easy due to differing finalties of protocols. Finalty is the amount of time it takes for a block to be permanently added to the chain.

- Hao et al. compare the performance of Ethereum's PoW consensus against PBFT on Hyperledger's Fabric platform [11]. The objective of the paper is to compare the performance of two popular and widely used consensus protocols. However, the selected protocols are different in the sense that Ethereum's PoW works in a permissionless setting whereas Fabric's PBFT works in a permissioned setting. The authors describe the process of deploying a blockchain application on both platforms and define some performance metrics for comparison. They used fairly standard metrics like average throughput and latency. The results indicate that PBFT is better than PoW in terms of throughput as well as latency. A noteworthy finding was that PBFT was only slightly better than PoW for a small number of transactions per second (workload), i.e. less than 100 tx/s. Beyond that, PBFT was far superior to PoW. This is an indication of the poor scalability of the lottery-based PoW consensus.

- In [12], Piriou et al. analyse the performance of the BizCoin cryptocurrency protocol (a vote-based protocol) in terms of consistency, and its ability to discard double spending attacks. The objectives of the paper are to define working consistency metrics and changes in the metrics were observed over time in a model built using the pyCATSHOO modelling framework. The authors propose three consistency indicators - consensus probability, the probability that all processes agree on the same blockchain state; consistency rate, the mean portion of the network that agrees on the most common blockchain state; and worst process delay, which is the length difference between the main blockchain

and its greatest common prefix[1]. Consensus probability was shown to gradually degrade from 1.0 to around 0.6 and remain constant thereafter. Consistency rate degraded only slightly and settled quickly at the 0.9 mark whereas worst process delay degraded exponentially from 0.0 to 0.5 and then remained constant. Using Markov chains, the authors analysed the probability of the blockchain being in a safe-state (no double spending). They plot this probability against the (stochastic) probability that a node is malicious. The resultant graph followed a sigmoidal curve against time.

- Asgaonkar et al. simulate a blockchain network with the objective of measuring the cost and throughput of the PoW consensus protocol [13]. One key difference between this paper and the others cited above, is that the authors model the protocol as a Poisson process. In a Poisson process events follow the Poisson distribution, i.e. they are independent, do not occur simultaneously, and their average rate of occurrence is constant. In this paper, the Poisson distribution determines when the peers sync their local blockchain copies with each other. Some key terms that the authors define are:

  - **0.5 chain** - Longest subchain that exists in the local blockchain of over 50% of the network peers.
  - **Throughput** - the length of the 0.5 chain.
  - **Orphaned block** - a block that has been proposed by a peer but does not appear in the longest chain of any peer.

  The authors plot the throughput and number of orphaned blocks against a varying rate of growth between the number of nodes and the average inter-sync rate parameter, λ. The rate of growth is represented as an equation between the two values, which signifies for instance, that λ increases quadratically, logarithmically or linearly with the number of nodes.

[1] A chain, $c1$, is the prefix of chain $c2$, if the last block of $c1$ is an ancestor of the last block of $c2$. For the greatest common prefix, $c2$ is the main-chain and $c1$ is the longest prefix for $c2$ that is present in every node's copy of the blockchain.

- In [14] Ampel et al. use Hyperledger Caliper, a performance benchmarking tool, to measure the performance characteristics of a blockchain application built on Hyperledger's Sawtooth platform. The objective of the paper is to use the Caliper platform to measure and analyze performance metrics of a blockchain built using the Sawtooth protocols - namely the Raft consensus protocol. The authors use metrics like throughput, latency, success rate of a transaction, and CPU and memory usage of the peers. The metrics are plotted against batch size (number of transactions per block) and the workload. Noteworthy findings indicate that throughput increases linearly with the batch size whereas latency increases exponentially with it. Latency also increases exponentially with an increasing workload while memory and CPU usage pick up almost exponentially at high workloads.

- Ahmad et al. compare five different protocols in [15] based on transaction throughput and latency. They measured these metrics while varying the number of network nodes. The protocols used were PoW, PoS, Proof-of-Elapsed Time (PoET), Clique (Proof-of-Authority) and PBFT. They found that Clique and PoS experienced the minimum latency, followed by PoET, PoW, and PBFT. In terms of throughput they found that upto 50 network nodes, Clique achieved the best throughput followed by PoET and PoS, but when nodes were increased beyond 50, Clique's throughput degraded. PBFT had a low throughput.

- In [16], Angelis et al. studied Aura and Clique - two variants of the Proof-of-Authority class of consensus algorithms; and classical PBFT, using the CAP (Consistency, Availability, Partition tolerance) theorem principles. The CAP theorem states that a distributed system cannot achieve consistency and availability when the network is partitioned in a way that messages may be arbitrarily lost. In a blockchain network, consistency refers to all nodes having the same blockchain copy, and availability refers to the network's ability to accept new transactions. Through a qualitative analysis, the authors show that Aura and Clique tend to prefer availability while PBFT prefers consistency.

2.3 Stochastic Modelling of Blockchain Systems

Stochastic modelling is the process of simulating a system whose next state is determined by its current state, some condition being fulfilled, and an element of non-determinism (i.e. randomness) that affects the state transition. For instance, a company's stock price on any given day, is determined almost entirely by its price on the previous day. However, some seemingly random factors (supply and demand) cause fluctuations in the price from day to day. Thus, it is very unlikely that the price will go from $50 today to $1000 tomorrow, irrespective of what the price was one month ago. It is more likely that the price fluctuates to the $40 or $60 mark which is closer to the current state of $50[2].

A similar tool - probabilistic modelling - is used to simulate a system where the transition from one state to another follows a probability distribution. However, the terms stochastic and probabilistic modelling are often used interchangeably. This is mainly because stochastic models use probability distributions to account for the non-determinism between state transitions. Figure 1 shows how a transition between two states (*inactive* and *active*) occurs based on some condition being fulfilled (the if-else block) and a probability distribution. More accurately, stochastic models simulate non-determinism using mathematical formalisms like automata and Markovian processes. These are discussed in more detail in Chapter 4.

Even basic Markov-chains can mimic non-deterministic network delays, packet errors, node failures and other stochasticities that occur naturally in blockchain networks. Such modelling techniques are already being used to predict outcomes in mechanical and physical systems. They have also been used successfully in the insurance industry to help insurers valuate assets, as well as evaluate the risk of disasters affecting an asset [17]. In this literature review, stochastic models were used to study characteristics of blockchain like security, performance, stability and scalability.

[2]Of course, this is a simple example and the stock market is often unpredictable.

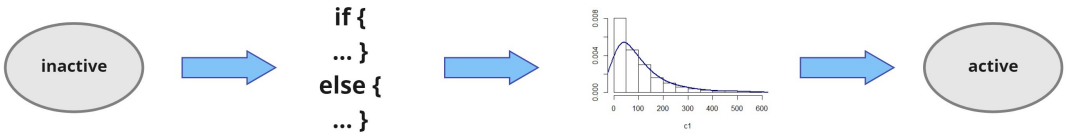

FIGURE 1: Stochastic modelling flow from inactive to active state.

- In [18], Duan et al. provide an overview of a formal verification process for blockchain systems. Their objective is to provide a replicable methodology for this process. This is achieved by designing a hierarchical and modular SDL (Specification and Description Language) model, for a private crowdfunding blockchain application. They use a modelling tool called Telelogic Tau for model verification. The focus of the research is on security and safety of blockchain systems. The authors outline general information and things to remember while building a blockchain model - this includes things like the contents of a block, how the consensus protocol fits into the model, and how to emulate malfunctioning nodes. They also provide formal descriptions wherever possible. Overall the methodology is fairly pliable and relevant to this research.

- The work by Gopalan et al. in [19], revolves around stability and scalability analysis of a blockchain system using modelling techniques. The objectives of the paper are to find a way to measure stability and scalability, and compare the results derived from a blockchain model versus a real deployment. The paper is highly technical and detailed. Stability is defined as the ability of a blockchain to be consistent across peers for short bursts of time, infinitely many times. Scalability is defined as the property of a blockchain network which is stable for a given burst-length as the number of peers increase monotonically. The authors study the blockchain as a directed acyclic graph (DAG). They define n-endedness as a property of the DAG where, on recording the path-to-root from each vertex[3], you will have a total of n different paths. Here, sub-paths are not considered different from their parents. They then show how *one*-endedness is

[3]The path-to-root is the sequence of vertices from a given vertex to the root of the DAG.

desirable as it relates directly to the network having no forks and consequently a successful consensus protocol. They found that as the block arrival rate (i.e. the rate of creation of new blocks) increases; the time to consistency increases monotonically, the consistency rate decreases almost linearly[4], and the consistency offset increases almost exponentially[5]. The experiments were conducted using simulation data as well as real data from the Bitcoin blockchain. For all experiments the results from the simulated and real data were comparable.

- Papadis et al. use modelling techniques in [20] to analyse block generation statistics of a blockchain system. Their objective was to compare the block generation statistics measured in a real blockchain application against those measured in a simulated model. They also analyse the impact of stochastic components on the probability of attacks on the network. The Ethereum testbed is used for building the real blockchain application, and the difficulty parameter is varied in the model as well as the Ethereum implementation. The difficulty parameter indicates how difficult it is to generate a new block at any given time. It is directly proportional to the time taken to generate a new block. Both experiments give comparable results. The authors use hashing power of nodes and network delays (block transfer delay and transaction processing delay) as the stochastic components in their experiment. Finally, they analyse the impact of delay and number of confirmations, on adversarial attacks. They found that the probability of a successful attack increases with increasing delay and decreases with higher number of transaction confirmations.

2.4 Key Takeaways

This chapter outlined the history of consensus protocols - from their use in distributed systems in the 1990's, to the post-Bitcoin explosion in new protocols since 2008. It

[4]Consistency rate is the fraction of the network agreeing on the same blockchain state.

[5]Consistency offset is the mean number of blocks that each node's local chain needs, to be consistent with the main-chain.

highlighted the importance of consensus protocols and research methodologies that have been used to study them. Finally, it gave a brief introduction to stochastic modelling and discussed ways in which modelling can be used to study blockchain systems. Some notable takeaways from the existing literature are:

1. Stochastic modelling techniques, as well as open-source blockchain platforms are prominent ways of studying a blockchain application (especially characteristics like performance and security).

2. Throughput and latency are the most commonly used performance indicators for blockchain applications. However, these are often accompanied by other metrics.

3. Stochastic models are a popular tool to analyse characteristics of blockchain applications.

4. Real blockchain deployments are used to verify findings from the models.

Despite their popularity in existing works, throughput and latency are not good enough performance indicators alone. For one, they tell us nothing about the consistency of local chains. Nor do they say anything about the number of invalid, or rejected blocks. Therefore, our experiments use throughput, latency, success rate, and the standard deviation of local chains to compare consensus algorithms. The success rate is taken as a ratio between the number of accepted blocks and the total number of blocks created (including ones that were rejected). The standard deviation metric is used to measure consistency amongst local blockchain copies. In addition, we consider two secondary metrics i.e. load tolerance and fault tolerance.

Using the CAP theorem as in [16], gives a different perspective on the characteristics of a protocol. However, this too is not enough alone. The authors of [16] suggest that their analysis can be backed up by implementing the scenarios described in their paper, and collecting measurements of metrics including throughput, latency and scalability metrics. Eventually, the CAP theorem can be used as a framework to analyze protocols, but metrics like throughput and latency are important to verify

the analysis. Finally, some of the most important findings were the tools used in the existing literature. Tools like Hyperledger Fabric and Ethereum - which are platforms to build blockchain applications - are mentioned above; with Hyperledger Caliper [21] - a blockchain performance measurement tool; and finally Modest [22], pyCATSHOO and Telelogic Tau - which are all stochastic modelling tools.

Although a lot was learnt from the literature review, the scope of this work is different from all the existing works. While most other research either compares two blockchain applications and their protocols, or compares the performance of a single application to its corresponding protocol's stochastic model, this work does both. Moreover, we use chaos engineering principles to test for fault tolerance - something that was not done in any of the existing literature. A comparison between this thesis and the existing literature is presented in Table 1 and the two prominent differences in our methodology are highlighted below.

1. Consensus protocols were carefully selected for this research, so as to cover a broad research area. The 4 protocols selected - PBFT, Tendermint, Clique and Raft - all belong to different protocol families. Chapter 3 discusses the taxonomy of consensus protocols and highlights the differences between the selected protocols.

2. In addition to primary performance metrics like throughput and latency, fault tolerance and load tolerance are also considered in this work. These performance indicators measure changes in the primary metrics when certain parameters are varied. Performance metrics are also discussed in Chapter 3.

This thesis lays down a framework for the comparison of blockchain platforms. It can be used to compare the performance and resilience of consensus protocols, and study the architectural differences between different permissioned blockchain platforms. Our methodology covers a number of alternative routes to performance measurement - like using Linux packages when the use of other open source software was not permitted by the platforms. Researchers can also use our methodology for building stochastic models, to build models of other popular consensus protocols. Through

TABLE 1: Comparing this work to the existing literature

Paper	Modelling	Application	Protocol Families	Performance Metrics	Load / Chaos Testing
[10]	Yes	No	PoW, BFT	L[1]	Load
[11]	No	Yes	PoW, BFT	TP[2], L	Load
[12]	Yes	No	NA (vote-based)	Consistency	Load
[13]	Yes	No	PoW	TP, OR[3]	None
[14]	No	Yes	Paxos	TP, L, SR[4], RU[5]	Load
[15]	No	Yes	PoW, BFT, PoS, PoA, PoET[6]	TP, L	Load
[16]	No	No	BFT, PoA	CAP Theorem	None
[19]	Yes	Yes	PoW	Scalability, Stability	Load
[20]	Yes	Yes	Ethereum (i.e. PoW or PoA)	Block generation	Load
This work	Yes	Yes	BFT, PoS, PoA, Paxos	TP, L, SR, Consistency	Both

[1] Latency; [2] Throughput; [3] Orphan Block Rate; [4] Success Rate; [5] Node Resource Utilization; [6] Proof-of-Elapsed Time

our experiments with stochastic modelling, we highlight the pros and cons of using stochastic models to evaluate the performance of consensus protocols. This can help other researchers determine whether or not to use stochastic models based on their own evaluation criteria. For instance, from our experiments we found that stochastic models were not accurate in their predictions of exact metric values i.e. the magnitude of results varies considerably between the stochastic models and blockchain applications. Therefore, using stochastic models to evaluate the performance of a single protocol (i.e. not a comparative analysis) may not be a good idea.

CHAPTER 3

Blockchain And Consensus Protocols

3.1 What Is Blockchain?

One way to look at blockchain is as a data-structure. Here, individual blocks are chained together, similarly to the nodes of a linked list, and each block is made up of two fields - transactions and header. These fields contain the list of transactions and other relevant information respectively. One piece of information stored in the header is the SHA-256 hash of the previous block. This previous hash is what creates the link between consecutive blocks. Figure 2 illustrates this. In practice, this data structure is used as a data-store, but is unique for its qualities of decentralization and immutability.

The blockchain data structure is usually used as a component in larger systems. Several other components interact with it by sending messages, and can either alter or query its state. The blockchain itself can interact with third-party software. This view of blockchain as a data structure will be useful in Chapter 4 which talks about blockchain-based applications. In this section, the focus is on immutability, decentralization and blockchain networks.

One of the defining attributes of blockchains is their immutability. Data once stored on them cannot be altered, updated or removed retroactively. This is because retroactive changes lead to a mismatch between the updated block's hash value and its hash value in the next block. To modify the blockchain state a network of peers

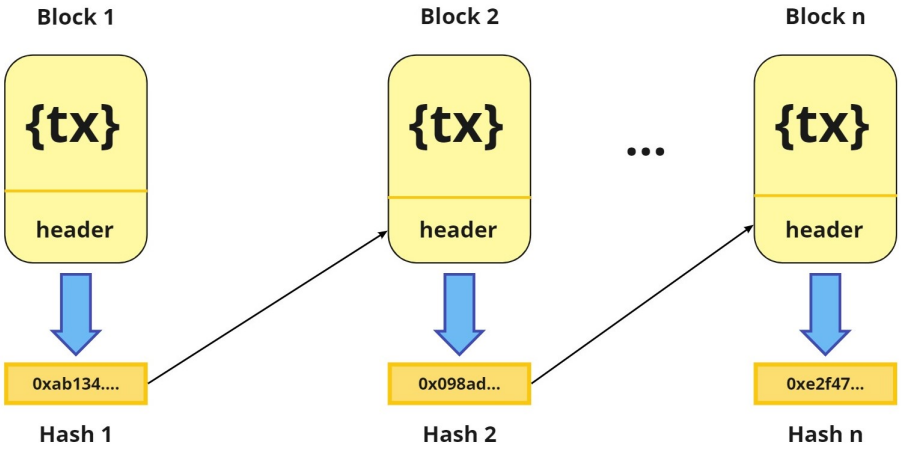

FIGURE 2: The blockchain data structure.

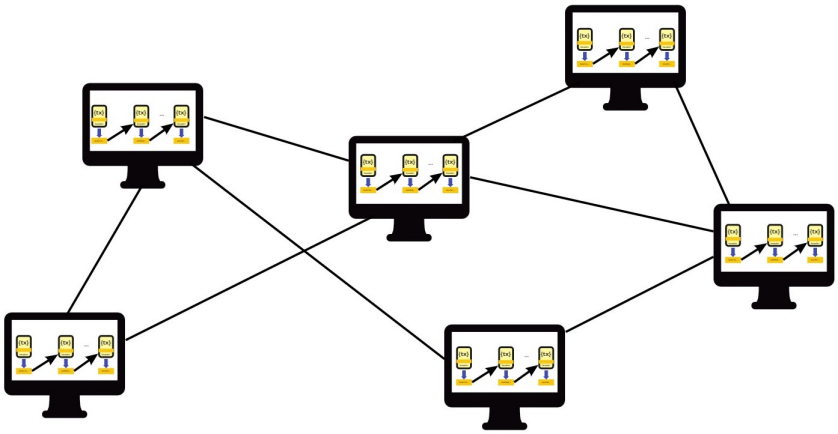

FIGURE 3: The blockchain network.

must first agree on the modification. It is important to note that a system does not have one single blockchain - the entire chain of data is stored at several nodes on a network (called the blockchain network). This network can be classified based on its access permissions into private, public or permissioned. A private network only allows authorized access to the data while public networks allow open access. Permissioned networks lie in-between and allocate specific permissions to peers[1]. Another thing to note is that different types of networks use different consensus protocols.

The second important feature of blockchains - decentralization - is the absence of a

[1]The words permissioned and private are often used interchangeably in relation to blockchain, as are the words permissionless and public.

central authority controlling the network. Most private and permissioned blockchain applications have at least a few authorized nodes on the network for openness and trust, while public blockchains are entirely decentralized. Hence, to redefine - blockchain is a permanent, distributed store of records that usually has no centralised authority.

3.2 Blockchain Consensus Protocols

Today, there are close to a hundred consensus protocols used in blockchain and distributed ledger systems [6]. As per the findings in a 2017 study [23], at the time of writing, 15 consensus protocols were used most commonly across several industries. Participants of this study include institutions like IBM, R3, Depository Trust and Clearing Corporation (DTCC), BigChainDB and banks like BBVA, UBS and more. There is no single best protocol as the choice depends on network structure, topology, desired confirmation times, security and other factors. Moreover, 36% of the study participants claimed to support or use pluggable consensus which allows you to create multiple chains on the same platform - each with its own consensus protocol. In this scenario each chain would use a single protocol at a time.

Most enterprises today, prefer private blockchain implementations like the ones offered by Hyperledger, Corda, Quorum, etc. because they feel comfortable having closed access to their data. Here, enterprise refers to any company, irrespective of size, that follows a centralised governance model (like a board of directors). They constitute a large majority of all corporations that exist today, while the opposing side is mostly made up of decentralised autonomous organizations (DAOs). Industries like healthcare and finance deal with a lot of sensitive user information - storing private data like this on a public blockchain would grant open access to it. Since the majority of corporations prefer private blockchains, focusing on consensus protocols used in these settings seems more relevant to the current state of the industry.

As mentioned, at the heart of every blockchain system is a ledger of transactions. Participants on the network make transactions and the ledger records them. But the ledger is more than just a data store - it functions as a state machine. It stores

not only the transaction history but also the global state of the entire system. This includes the balances of all accounts in the network; the amount of funds present at every address; and in the case of smart contracts, the last known values for every internal variable that exists. Following from the concept of state machine replication (SMR), a blockchain system can be conceptualised as a machine whose state changes deterministically with time. Given some valid transactions, the peers will perform predefined computations which change the state of the blockchain. If all nodes receive the same input transactions and can agree on their order (while overcoming node failures and transmission errors), consensus can be reached and all local copies of the chain will be consistent.

According to [9], to reach consensus in distributed systems, an important step is to find ways to communicate efficiently. In practice, this is done using atomic broadcasts. Using simple broadcast messages ensures that all participating nodes receive all of the input messages. Atomic broadcasts further ensure that these input messages are received in the exact same order by each peer. This allows the nodes to reach consensus individually. There are four central properties of atomic broadcasts:

- ***Validity***: If a message is sent by a valid node, it will be included in the consensus process.
- ***Agreement***: If a message is delivered to a valid node, it will be delivered to all valid nodes.
- ***Integrity***: Each valid node can broadcast a given message only once.
- ***Total Order***: All nodes must agree on the order of the messages.

This leads us to the properties of consensus protocols and the systems that use them. Two of the main properties of distributed consensus protocols are *safety* and *liveness*.

The safety of a consensus protocol is concerned with a system never reaching an undesirable (bad) state and liveness is concerned with the system eventually reaching a desirable (good) state. In simple terms, the safety property defines what must not

happen, while liveness defines what must eventually happen. Another way of looking at this is that the safety property must hold from the system's beginning to its end of life, whereas liveness becomes true at some point in the future (either once or several times).

These properties help define the correctness of a distributed consensus algorithm. Looking at the conditions below, it is no surprise that atomic broadcasts work well for consensus protocols. The conditions of correctness are as follows:

- ***Validity***: Any value decided upon must be proposed by at least one of the processes.
- ***Agreement***: All honest processes must agree on the same value.
- ***Termination***: All honest nodes must eventually decide on some value.

The validity and agreement conditions relate to the property of safety because honest nodes will never agree on random, trivial, or different values. Further, these conditions must hold from the beginning to the end. Termination on the other hand, relates to liveness, because a decision must eventually be reached. This condition must be met recurrently sometime in the future.

In a blockchain context, the definition of safety is often given as the combined definitions of validity and agreement as described above. Similarly, liveness is often defined by the termination condition. Ideally, most systems should be able to easily provide both safety and liveness as defined in this modern sense. Practically, however, we must also account for malfunctioning nodes and network trouble. If we add in the condition of fault tolerance, a system cannot achieve all three of safety, liveness and fault tolerance. This is called the Fishcer-Lynn-Patterson (FLP) Impossibility.

The FLP Impossibility is one of two trilemmas in the consensus protocol domain. The second of these is called the CAP Theorem. To understand the CAP theorem it is first important to understand three properties of systems that use consensus protocols.

1. ***Consistency***: This property holds true when all peers produce the same valid

output. In other words, the system is said to be consistent when each peer's local state is consistent with every other peer's local state.

2. ***Availability***: This property holds true when every node has constant read and write access to the system i.e. the system is not down.

3. ***Partition Tolerance***: A system is said to be partition tolerant if it runs normally even when the network is partitioned such that two or more nodes are unable to communicate with each other.

This gives us the CAP Theorem, which states that no consensus protocol can achieve consistency and availability when the network is partitioned in a way that messages may be arbitrarily lost. Consistency and partition tolerance are related to the safety property because they both deal with validity of the decision and agreement between nodes. Further, they must always be true. Availability on the other hand, is closely related to liveness. This is because if a system never reaches consensus (no termination), it will never be available. The wording of the theorem hints that most reasonable systems assume the occurrence of partitions. Therefore, the trade-off is between consistency and availability. Finally, it is important to note that none of these three properties are absolute - rather, they belong on a spectrum.

By far, the two most common ways to classify consensus protocols are with respect to their fault tolerance or with respect to the incentives they offer. In the case of fault tolerance, a consensus protocol can either be crash fault tolerant (CFT) - where it can tolerate node failures; or Byzantine fault tolerant (BFT) - where it can tolerate byzantine as well as faulty nodes. In relation to incentives, consensus protocols may be incentivised - where they reward the block creator with some token; or non-incentivised - where there is no reward for block creation. Generally, incentivised protocols are used in the public setting whereas non-incentivised protocols are used with permissioned or private blockchains. Therefore, this research deals with non-incentivised protocols.

3.2.1 A Taxonomy of Consensus Protocols

While selecting consensus protocols to use for the experiments, a taxonomy of consensus protocols was used to understand and recognize the classifications of consensus protocols. The authors of [9] provide this taxonomy and present four types of properties – structural properties, blocks and rewards, security, and performance. The structural and performance properties of consensus protocols are most relevant to this research and are discussed below.

Structural properties of consensus algorithms can be divided further into the following subcategories:

1. **Node type** - depending on the platform used, a consensus algorithm may have to deal with multiple node types - like full nodes (that store the entire blockchain locally), validator nodes, endorsers (which only validate transactions) and light clients (which verify new blocks without storing the entire blockchain locally)[2].

2. **Structure type** – Consensus protocols can use single or multiple groups (committees) to reach consensus. Therefore, they may either have a single committee of validators which generates the next block, or multiple committees that work independently to generate the next block. Each type of committee must account for some further considerations. For example, a single committee can be open or closed to new members, it can be static or dynamically changing its members, and have implicit or explicit formation rules. PBFT is one example of a protocol with this structure. Similarly, a multiple committee mechanism must select an overall topology (i.e. flat or hierarchical) amongst the committees, and also decide whether membership is static or dynamic. In the Raft consensus protocol, if the network is partitioned, a single committee is broken into multiple flat committees. If any partition of the network contains more than two-thirds of the participating nodes, this partition becomes the main committee and the others must follow its decisions (hierarchical topology).

[2]The examples given are not all found together, they show up in different platforms.

3. **Underlying mechanism** – This refers to the core method of reaching consensus and can roughly be classified as either a *lottery-based* (PoW and PoS), *vote-based* (Tendermint) or *coin-age-based* mechanism.

Performance properties include throughput, latency, fault tolerance, scalability and energy consumption. These properties (or metrics) are discussed in more detail in the next section. The block and reward properties from the taxonomy are relevant for incentivised consensus protocols and are not discussed here. The security properties include authentication requirements, non-repudiation and censorship resistance of the protocols. They also include adversary tolerance and tolerance to attacks like Denial of Service (DoS) and Sybil attacks. Although the security of a system and its performance are related, the properties outlined do not provide useful information about performance of the protocols. For this reason they are mostly left out in this research. Moreover, they are harder to simulate and measure.

Below, the selected consensus protocols for this research are examined. They broadly cover the prominent families of non-incentivised consensus algorithms that have emerged in the industry over recent years. These are - byzaninte fault tolerance (BFT) based, proof-of-stake (PoS) based, proof-of-authority (PoA) based and Paxos-based. Before jumping into the selected protocols, it may be helpful to describe briefly these four families.

BFT-based protocols are always byzantine fault tolerant. Usually, they follow multiple rounds of voting to achieve consensus - similar to PBFT - but this is not necessary. Most BFT-based protocols suggest improvements over PBFT. For instance, FastBFT reduces the number of voting rounds and improves performance without compromising security. The second family, PoS-based protocols, use a proof-of-stake model somewhere in the consensus mechanism. It is commonly used for leader election where, for instance, the block proposer (leader) for the next round is decided based on each validator's stake in the system. Many modern PoS projects separate the "stake token" from their main token. This means that the token used to represent your stake in the system is different from the token you would use to interact with applications on the system. This is also the case with Tendermint. Interestingly, Tendermint is

a DPoS-BFT protocol - it uses a PoS model for leader election and voting rounds to commit blocks. DPoS stands for Delegated PoS, a variant of PoS where network participants can delegate their tokens to certain validators as a vote of confidence.

The PoA protocols are a popular class of non-incentivised protocols. They store proof of the validators' identities to ensure that nobody in the validator set is byzantine or malicious. In case a validator is malicious, the proof of identity can be used to take appropriate action. While these protocols are also byzantine fault tolerant, they can reach much better performance than BFT-based protocols due to lighter message exchanges. They are best used in low-risk scenarios, where there is a great deal of trust in the validator set. For example, three of Ethereum's testnets use PoA protocols - one of which is Clique (Rinkeby and Görli testnets). This is a low-risk scenario because none of the tokens on the testnet have real monetary value, and also the validator nodes on the testnets are run by trusted members of the Ethereum community.

Finally, Paxos-based protocols provide algorithmic or understandability improvements over the Paxos protocol proposed by Lamport. Raft is a popular Paxos-based protocol which, like Paxos itself, is not byzantine fault tolerant. It is not entirely clear what protocol family Raft belongs to however - some sources claim that it is a proof-of-capacity protocol. However, given its close association to the Paxos protocol (and seeing how Paxos and Raft always seem to be mentioned together while discussing blockchain consensus protocols), Paxos-based seems a more apt classification. Other Paxos-based protocols are used in other fields of computer science that require consensus - like database management and state machine replication. In general any implementation of the Paxos protocol can be classified as a Paxos-based consensus protocol. Since Lamport did not provide a detailed implementation for Paxos, many early variants of Paxos emerged as people implemented the protocol for their own use-cases.

3.2.2 Selected Protocols

This section describes the selected consensus algorithms and provides a summary based on the taxonomy described above. The pseudocode for each algorithm can be found in Appendix B.

1. Practical Byzantine Fault Tolerance (PBFT)

Practical Byzantine Fault Tolerance was the first practical implementation of a byzantine fault tolerant consensus protocol. It was proposed before the bitcoin revolution and thus was used in distributed systems before blockchain systems. The basic algorithm however, follows a similar procedure in both. It consists of the three-step process of: pre-preparation of a block, preparation by collating peer votes, and committing blocks that have received a majority vote of acceptance. The full algorithm is outlined below.[3]

1. First, a leader is selected from amongst the network peers. This is done in a round-robin fashion.

2. Once elected, the leader validates a group of transactions and creates the new block.

3. The leader will broadcast this block to all the other nodes with the "*preprepare*" message.

4. In the first phase, the validating peers receive the block, check its validity, validate its transactions, and if everything checks out, they broadcast the "*prepare*" message and start the second phase.

5. If a peer receives the "*prepare*" message from more than two-thirds of the network (minus the leader), it will broadcast the "*commit*" message to start the final phase of voting.

[3]In the following sections, the terms 'peer' and 'node' are used interchangeably to refer to the nodes on the blockchain network.

6. In the final phase, when a peer receives a "*commit*" message from more than two-thirds of the network, the block is added to the chain.

While the peers are voting on the current block, the leader can create the next block and broadcast it to the network simultaneously. The peers therefore vote on multiple blocks at the same time, but each block is in a different voting phase. The entire voting process for a single block can continue for multiple rounds (where each round consists of the 3 phases) till consensus is reached.

PBFT includes a timeout period for rounds. If no consensus is reached within this timeout period, the round ends without a committed block and the leader is changed in the next round. If the voting ends within the given time, the leader will remain the same in the next round.

2. Tendermint

Tendermint is a delegated proof-of-stake byzantine fault tolerant (DPoS BFT) protocol. This means that it considers each user's stake in the network and can handle byzantine faults. It is a round based protocol, where each round consists of four steps – *propose, prevote, precommit* and *commit*.

1. First, the leader (or proposer) is selected in a weighted round-robin fashion. Here, the weight is decided by the peer's stake in the system.

2. Once elected, the proposer must broadcast a block to the network. If it had locked onto a block in the previous round, it can send that block along with a proof-of-lock (explained below) or else it can create a new block.

3. The proposer then sends the block proposal to its neighbours with the "*propose*" message.

4. The neighbours in turn pass the message on to their neighbours using a gossiping protocol.

5. Each node that receives a "*propose*" message, sends a "*prevote*" message across the network.

6. At this point, any node that has received the block proposal as well as a "*prevote*" message from more than two-thirds of the network must itself broadcast a "*prevote*" message if it hasn't already.

7. If any validator node receives more than 2/3 "*prevote*" messages and has sent one itself, it will now send a "*precommit*" message.

8. At this point, the validator node will lock onto the block that it is pre-committing and compile a proof-of-lock by collecting the 2/3 majority pre-votes it has received.

9. If a node receives more than 2/3 "*precommit*" messages, it will add the block to its local blockchain.

The usefulness of the locking mechanism is that since the locked block has reached the "*precommit*" stage, it has already been vetted by the network. If the locking node is then elected leader in the next round, it will use this locked block instead of creating a new one. The lock can be lifted in two circumstances - if that block is committed, or if a new block is available to lock[4].

In the Tendermint protocol, message passing is done using peer-to-peer gossiping i.e. peers only communicate with their neighbours. There are no forks while using Tendermint - the protocol lays more emphasis on consistency than on availability. In other words, it focuses on having a consistent local chain across the network rather than being able to process every transaction that comes through. This means that although the chain will never fork, the system may be down more often. Tendermint manages to avoid forking by allowing validators to sync their local blockchain copies. However, this is not a step in the protocol's algorithm but is instead done by the Tendermint Core implementation of the protocol. A major point of difference between

[4]If a node receives a proof-of-lock for block R' and it already has a lock on block R, such that $R < R'$, the node must release R and lock onto R'.

Tendermint and PBFT, is that in PBFT a timeout results in a new leader whereas in Tendermint the leader is changed at the end of each round regardless.

3. Clique

Clique is a Proof-of-Authority (PoA) protocol where the validators (called signers) are authorised nodes whose identities are pre-verified by the network. The word clique literally means "inner circle". In each round, one signer is elected as the leader, who along with a few other nodes is allowed to propose a block, while the majority of the nodes must validate the block. The leader is elected in a round-robin fashion. After every 30,000 blocks (or one epoch), an empty block is appended to the blockchain. The algorithm is as follows:

1. The leader collects transactions and creates a block by solving a hash puzzle (an easy problem that takes less than about 15s to solve).
2. The leader signs the block (called sealing) and broadcasts it to the other signers.
3. When a signer receives a sealed block from the leader, it will validate and add the block to its blockchain.
4. After a block-period of about 15s, the leader changes and the next round begins.

Importantly, a signer is only allowed to seal a new block after every x blocks, where

$$x = floor(total\_signers \div 2) + 1$$

Consequently, at any given time there are only $(total\_signers - x)$ potential block proposers in the network. If a node other than the leader for the current round proposes a block, the hash difficulty is brought down to 1 (default is 2). To deal with forks, the chain with the largest cumulative difficulty is always preferred. When the elected leader node crashes, the network waits for a period of $(signer\_num \times 500)$ seconds[5]. Whichever node's turn it is to become the leader after this period will be the new leader.

[5] $signer\_num$ is the position of the signer in the round-robin order.

In case a node that is not allowed to propose a block does so (belongs to x as discussed above), the other nodes can vote to drop this out-of-turn node from the clique. To add or drop a signer, the leader can propose to add or drop the peer by using their unique identifier (i.e. address or signer_uid). This proposal is sent along with the sealed block. The other signers can vote on this proposal whenever they seal their own block. At any instant if a majority add/drop vote has been reached, action can be taken immediately. Generally, at the end of each epoch, existing votes are tallied, changes are made to the clique and vote counts are reset.

4. Raft

Raft was designed to be an understandable and practical protocol. It is similar to Paxos in terms of results and efficiency, but utilizes a different structure. Over the course of the protocol, each node can be in one of three states – follower, candidate, leader. Raft consists of two main sub-problems - leader election and log replication. Leader election is the process of deciding who proposes each new block and log replication must occur across the validator network. The algorithm listed below gives steps for each of these two sub-problems. It can also be read as one round of the protocol starting from when the current leader is dismissed.

Leader Election

1. If a node does not receive a heartbeat signal from the leader within a timeout period (randomised per node between 150-300ms), it becomes a candidate node and starts a new election process by giving itself one vote.

2. The candidate(s) broadcast a "*request vote*" message to their peers.

3. Any node that receives the "*request vote*" message and has not already voted, sends its vote to the candidate it first received the message from. Subsequently, it resets its timeout.

4. The election repeats till a majority leader is elected.

Log Replication

1. The leader sends a heartbeat signal (empty log update with an "*append entries*" message) to all validator nodes to show that it's alive.
2. Nodes respond when they receive the "*append entries*" message.
3. All new transaction requests arrive at the leader node (either directly or indirectly through other nodes).
4. The leader validates transactions and appends them to its log.
5. The leader then sends a non-empty log update with the next heartbeat signal.
6. Nodes reply to the "*append entries*" message.
7. If the leader receives a 2/3 majority reply from the followers, it commits the changes and broadcasts this information so the followers can also commit.
8. After committing, the leader sends an acknowledgment to the client.

The protocol works like this assuming no network partitions occur. In case of network partitions the network will become a multiple committee network. Here, each partition will function as an independent network but a partition only commits changes if it consists of the required two-third majority of peers.

Table 2 summarizes the four protocols described above on the basis of the taxonomy provided in [9], the protocol family each of them belongs to, and the platforms that offer the protocols.

TABLE 2: Comparing the selected consensus protocols

Protocol	Family	Platform	Fault Tolerance	Structure	Underlying Mechanism
PBFT	BFT-based	Hyperledger Sawtooth	BFT	Single Committee	Vote-based
Tendermint	PoS-based (DPoS-BFT)	Tendermint Core, Cosmos SDK	BFT	Single Committee	Vote-based
Clique	PoA-based	Ethereum's Rinkeby testnet	BFT	Single Committee	Leader-follower
Raft	Paxos-based	Hyperledger Fabric	CFT	Single / Multiple Committee	Vote-based

3.3 Performance Measurement

Quality assurance and quality control are critical steps in the software engineering lifecycle. Service providers and development teams agree to a service level agreement (SLA) with customers, which defines the level of quality the customer should expect. Consequently, software quality assurance (SQA) entails a set of processes that must be followed in order to achieve an end-product of acceptable quality; and quality control (QC) - done through software testing - is used to ensure that software is indeed of the necessary quality.

Performance measurement is a key part of the software testing process. It allows the product's quality to be measured and fine-tuned till it reaches the specified standard. This ensures that the SLA is satisfied before the product can be released to the public. Since quality is subjective, measuring it is a demanding task. Testing helps in setting realistic expectations.

In software engineering it is not uncommon to make design choices based on system performance. In [24], the authors suggest testing early in the development lifecycle. This allows development teams to change architectural decisions while they still can and improve product quality. They found that a large number of performance issues come down to architectural decisions made early on - like the choice of middleware in distributed systems, or in our case, the choice of blockchain consensus protocol. In blockchain systems, better performance intuitively means lesser processing times and faster transaction confirmations - consensus protocols have an effect on both.

A survey conducted by the authors in [25] highlights nicely the importance of performance as a software quality indicator - using P2P money-lending applications as the subject of the survey. The authors collected and analysed public opinion of 18 such mobile applications, with the objective of detecting key drivers of user satisfaction in digital lending apps. Public opinion was gathered with the help of user reviews and unstructured interviews. Using sentiment analysis tools, the authors ranked 15 top drivers of user satisfaction in money-lending applications. Many top drivers - responsiveness, reliability, accuracy and app performance - are partially or

directly related to performance of the apps. Some other drivers include ease-of-use, credibility, user-incentives, etc. Although the study has its shortcomings - data is limited to the Indonesian app store, and the study was exclusively concerned with mobile applications - it gives an idea of what end-users deem important in today's fintech applications, and performance tops the list.

Combined with the popularity of blockchain in the finance sector, we can begin to make a case for the importance of performance measurement in blockchain applications. Here, the focus is on the performance of blockchain consensus protocols.

- The performance of consensus protocols is largely representative of the performance of the entire blockchain system.
- Performance testing can help detect failures or anomalies in the protocol. For instance, a very low throughput or very high latency for block creation could indicate a shortcoming or error in the protocol.
- It can reveal inefficiencies; and especially while developing new protocols, it can help assess their progress and understand them better.
- It helps to gain a better understanding of the workloads that a protocol can handle effectively. Without some form of performance testing under load, the system could possibly crash or considerably slow down in production.
- It offers a way to compare protocols (or applications for that matter) against each other.

The authors of [26] define two classes of performance testing in distributed systems - external and internal testing. External measurements measure performance at a macroscopic level; like the number of read/write requests in a distributed database system. Internal performance measurements work at a microscopic level, dealing with the performance of individual components of a system. In blockchain and distributed systems, consensus protocols are responsible for reaching consensus system-wide. However, since consensus is just one component of a blockchain-based application, measuring performance of blockchain consensus protocols could fall in either

of the two categories - external or internal testing. But, how do we actually measure performance?

3.3.1 Performance Metrics

A metric is a standard of measurement - they are the criteria which help us measure performance. For instance speed and responsiveness can give us information about how good a machine is. Here, speed and responsiveness are the metrics - they help better understand the health of a system, its computation capabilities, and memory and network characteristics. In the case of consensus protocols, some useful metrics are throughput, time for consensus to be reached, and energy consumption of the protocol. In distributed systems, performance metrics can be used to describe performance at different levels - individual nodes, groups of nodes or the system as a whole. This is in line with the two classes (internal and external) of distributed system performance testing mentioned above. The levels mentioned in [27] and [26] are:

1. **System level**: This is the highest level at which performance can be measured. It deals with the performance of the system as a whole and can be fairly complicated to measure.

2. **Cluster level or Service level**: This is concerned with the performance of components that work together in a group or provide a specific service. In [27] this is called the distribution-unit cluster level.

3. **Machine level**: At this level, the performance of a single machine or node is measured.

4. **Process level**: At this level, the performance of a single process is measured. Due to a lack of attention in recent literature it would be safe to either omit, or club this with the distribution-unit level - where we measure performance of several processes that share the same memory.

Consensus protocols themselves fall under the cluster or service level category - the consensus process is carried out by some group of nodes which loosely include miners, validators and orderers. However, since most rounds of consensus end with block generation (which is reflected at all nodes) the effects can be felt at the system level. Therefore, the metrics used in this work fall in these two categories - system-level and service-level.

3.3.2 Selected Metrics

As discussed, performance metrics are a way to quantify the performance of a system. Two types of metrics have been selected for our experiments - primary and secondary metrics. Primary metrics measure certain aspects of the system directly. These include throughput, latency and success rate. Secondary metrics on the other hand measure changes in the primary metrics when certain attributes of the system are changed. For example, changing the number of nodes in the blockchain network would have some effect on the latency of the protocol. This secondary metric could be called scalability. The selected metrics are described in more detail below.

Primary Metrics

1. **Write Throughput** - Defined as the number of transactions added to the blockchain per second.

$$TP = \frac{(total\ transactions\ added\ to\ chain)}{(total\ runtime)}$$

2. **Average Write Latency** - The amount of time it takes for a transaction to appear on the blockchain, from when it was made. We are concerned with the average over all transactions.

$$L = \frac{\sum_{tx=1}^{TX_{tot}} (T_{txCommitted} - T_{txCreated})}{TX_{tot}}$$

where TX_{tot} is the total number of transactions,

$T_{txCommitted}$ is the timestamp when a given transaction is committed,

$T_{txCreated}$ is the timestamp when a given transaction is created by the user

3. **Success Rate** - The ratio of the number of blocks successfully added to the blockchain to the total number of blocks created (includes invalid and orphan blocks).

$$SR = \frac{(total\ successfully\ added\ blocks)}{(total\ blocks\ created)}$$

4. **Std. Deviation of Local Chain Lengths** - The standard deviation of the lengths of each node's local blockchain copy. It was found to be more useful than success rate while studying the secondary metrics in the stochastic models.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

where N is the total number of nodes in the blockchain network,

x_i is the length of the blockchain at node i,

μ is the mean blockchain length for all nodes in the network.

Secondary Metrics

1. **Load Tolerance** - It is measured by observing changes in performance under a varying input transaction load.

2. **Fault Tolerance** - It is measured by observing changes in performance when different types of faults appear in the network (crash faults, omission faults, byzantine faults, etc.).

This chapter went over some important decisions made for this research. It outlined the choice of protocols and their metrics and also provided supporting arguments from the related literature. The next chapter looks at techniques used to measure these metrics.

CHAPTER 4

Blockchain Platforms and Modelling

Blockchains rely heavily on the structure of their underlying protocols. Many protocols include some form of voting in their procedure, but most are distinct and unique. The challenge for blockchain is maintaining trust on the network, along with decentralisation and security. The network, its participants and their resources, constitute a blockchain system. This chapter discusses an approach to build models of these systems, build the systems themselves, and measure their performance. Finally, the platforms used to build the blockchain applications are compared.

4.1 Blockchain Systems

This section gives a brief background on the platforms used to build our blockchain applications and how they use the blockchain data structure. Generally, it is incorporated into the application's backend or becomes a part of its basic functioning.

The authors of [11] and [28] codify a blockchain system into four abstract layers – the data-model, consensus, execution, and application layers. The data-model layer specifies the data structures and data types of each block; the consensus layer deals with creating blocks by reaching consensus on the network; and the execution layer enables smart contracts and their interaction with the blockchain data structure. For instance, the Ethereum virtual machine (EVM) is used to interpret methods written in Ethereum smart contracts, whereas Hyperledger Fabric's runtime environment

performs similar functions to interpret chaincode. Unlike Ethereum smart contracts, which are written in Solidity, chaincode can be written in a number of programming languages including Go, Python and Javascript. The topmost layer houses the entire application with its business logic, which interacts with the blockchain through the execution layer. Decentralized applications (DApps) and decentralized autonomous organizations (DAOs) usually live in this layer.

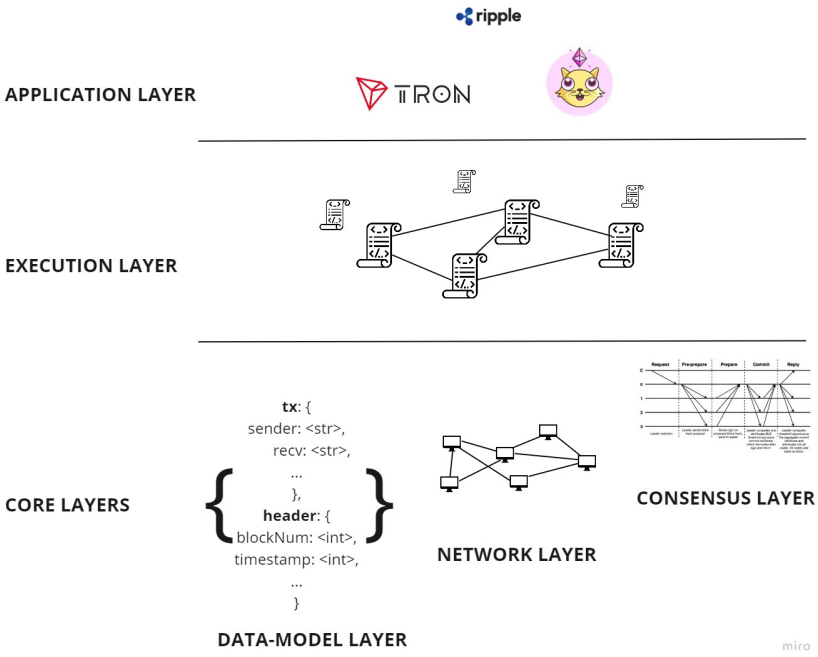

FIGURE 4: Layers of a blockchain system

Another view of blockchain systems shown in Fig. 4, adds a separate network layer to govern the peer-to-peer network and its communication protocols. The network layer is important because it is sufficiently distinct from the consensus layer. However, the data-model, consensus and network layers are tightly linked - finding agreement on the blockchain state (data-model layer) requires communication amongst the nodes (network layer). The effect of this interconnection is that changes in the network and data-model layers affect the consensus layer. For instance, block size, network errors and faulty nodes can interfere with the process of consensus. However, modern tools and protocols are built to overcome, minimize or work with this variability.

4.1.1 Blockchain Platforms

There are many popular blockchain/DLT platforms that enable the development of blockchain applications. For instance, Ripple is a prominent cross-border payment corporation which also offers DLT services, and BigChainDB uses the blockchain data structure as a distributed, immutable database. Quorum, Corda, OpenChain are some other examples. One major player is Hyperledger. Established under the Linux Foundation, Hyperledger is a suite of blockchain technologies offering frameworks, libraries, and tools to be used with blockchain. These include monitoring tools, a digital credentials tool, and operational tools (think AWS for blockchain). All of these platforms provide technology stacks to help developers build blockchain applications.

The concept of smart contracts, i.e. executable code stored on the blockchain, was first introduced in the Ethereum yellow paper [29]. The two main contributions of Ethereum are the EVM, which is embedded within each node, and an object-oriented programming language called Solidity, which is used to write smart contracts. Together, they standardize the development process of DApps hosted on the blockchain.

Simply put, smart contracts are scripts on the blockchain that execute specific procedures when invoked by an external address. They can be invoked by addressing transactions to the contract's address. Smart contracts allow developers to emulate intricate financial tools like loans and currency exchanges in a decentralised manner. Each smart contract goes through some stages in its lifecycle, which are loosely defined in [30].

1. **Negotiation and Formation** - All parties involved form an agreement about the contract's function and translate this into code. Consider a hypothetical example, where a client promises to pay the full amount for a car in monthly installments by a certain date. According to the contract, if the dealership receives the money on time, proof of ownership will transfer to the client and will be stored on the blockchain permanently. If not, ownership will remain with the dealership and any money will be returned to the client.

2. **Storage and Notarization** - Contracts are stored on the blockchain with a unique address so they can be executed. Since they are cryptographically signed by all involved parties they are automatically notarized, but they must be attested by the network as well. In the dealership example, the contract is signed using each user's secret key and attested by the network.

3. **Execution** - Contracts can be executed by sending a transaction to their address. On the Ethereum blockchain, these transactions require some gas, which has an associated monetary value. This "usage cost" prevents the overuse of compute resources, and forces developers to use optimal logic and to be mindful of the cost of each operation. Monthly installments can now be sent to the dealership contract and can tracked by it.

4. **Monitoring and Enforcement** - Once deployed, a contract's execution can be interfered with, leading to security vulnerabilities. It is important to include validation checks in the code as a precaution measure and to monitor the usage of a smart contract. Relevant software design patterns can be used to implement these checks.

5. **Termination** - Once the contract's conditions have been met the contract must be terminated. Terminating a contract clears its internal state, and makes the contract unusable in the future. Not only does this save memory but also provides security.

Today, the Ethereum ecosystem has expanded into three networks - the main or public network, a few test networks, and private networks. The main and private networks use real money whereas test networks use pretend money. Private networks are completely independent networks forked from the main network. The forked chain's protocols can be modified and it can be used privately. One popular example of this is JP Morgan's Quorum. All Ethereum projects in production live on the main network. The test networks, like Rinkeby and Ropsten, are used to test DApps and provide different consensus protocols from the main network. Rinkeby uses the Clique protocol.

Each shortlisted consensus protocol is available on a different platform, which despite their differences, have a lot in common. Below is a brief overview of the four platforms used in this research.

Hyperledger Sawtooth

Sawtooth [31] uses a modular framework, which separates the system's core logic from application-level procedures. This makes it easier for developers to work with, while allowing them to build as simple or complex applications as they need to. However, it works well only with Ubuntu (18.04) Bionic. One thing that separates Sawtooth from the other platforms on this list, is its support for dynamic consensus. This means that Sawtooth allows switching between consensus protocols in-between voting rounds. It provides Go, Java, JavaScript and Python SDKs. We used version number 1.2.6 for our experiments.

Cosmos SDK

Comos [32] is an open-source framework that supports PoS and PoA blockchain applications. Its selling point is interoperability amongst its PoS blockchains, through the Cosmos Hub. Cosmos Hub is the collection of blockchains that run different proof-of-stake applications using the cosmos-sdk. Cosmos runs through Tendermint Core - which uses the Tendermint consensus protocol, and comes with an application-blockchain interface (ABCI) to communicate with external applications (like the ones built with Cosmos). Cosmos allows developers to build applications of varying complexity, which can interact with the blockchain in different ways. We used Cosmos Launchpad (v0.39) for our experiments. The transaction flow in a typical Cosmos app is as follows:

1. The client sends a message to the Cosmos app through the CLI or HTTP requests.

2. Based on the message type (make transaction, query blockchain, etc) a message object is created.

3. Object creation triggers an event, which is handled by a handler. The handler calls appropriate functions to handle each event, and each function usually ends with a call to the keeper.

4. The keeper is the only component of the application that communicates directly with the blockchain. Based on the handler's instructions, the keeper can either read from or write to the blockchain.

Go Ethereum (geth)

Go Ethereum [33] is an Ethereum client written in the Go programming language. Like other Ethereum clients, it resides on each node and can run on the main network as well as some of the test networks. Consequently geth offers the Ethash protocol (Ethereum's PoW), IBFT [34] and one version of PoA (Clique). Geth does not offer any SDK, but includes a JSON-RPC API, and can be used alongside a variety of libraries written in other programming languages, to run, maintain, debug, and monitor nodes on an Ethereum network. Geth v1.10.3 was used to build the application in this research.

Hyperledger Fabric

Fabric [35] is a permissioned DLT platform which can be used to build DLT applications for production. The project's architecture is modular and highly configurable. It allows chaincode to be written in Java, Go, JavaScript or Python and currently supports the Raft and Kafka consensus protocols. The ledger of transactions on Fabric is shared amongst multiple organizations, each having a number of peer machines and user accounts. Fabric 2.x was used for our experiments. The transaction flow for a typical Fabric app is as follows:

1. The client sends a transaction to every organization.

2. An endorsing peer from each organization validates the transaction, and if valid, sends back an endorsement to the client.

3. The client collects the endorsements and sends them to an orderer organization, which runs the ordering service, i.e. the consensus protocol.

4. Provided a majority of the organizations have endorsed the transaction, it is accepted by the orderers and included in the ordering process.

5. Once ordered, a batch of transactions are sent to each organization where they are committed to the organization's ledger by its peers.

4.2 Stochastic Modelling

Stochastic modelling is the process of modelling under probabilistic uncertainty. In other words, it is used to model processes that contain randomness in output determination. This means that the relationship between input and output variables of stochastic systems is not deterministic, and probability distributions can be used to account for this variability. Before getting into the technical details of stochastic modelling this section answers the questions: why modelling? What are the benefits of using models over the blockchain platforms discussed above? And what performance metrics can be measured using models?

Chapter 3 spoke about early testing of software systems. To early-test a blockchain application, one must deploy an entire network, build the application, and then analyse its performance. This is where modelling techniques can be useful. Using models is much more efficient because they require neither network nor application - all they need is the consensus algorithm. Another advantage of modelling techniques is that they are configurable and reusable. For example, since many protocols require the ability to count votes, certain functions can be reused by models of different protocols. Their configurable nature also allows a large number of metrics to be defined.

Chapter 2, illustrates many use-cases of stochastic modelling, mainly for measuring characteristics of blockchain systems like performance, security, and stability. These models used metrics like rate of growth of the blockchain, average commit time, probability of forks, and the rate of invalid blocks. Stochastic models are a popular

choice because they can model real-world communication scenarios well - which are central to blockchain applications. Since stochastic modelling can account for non-determinism in state transitions, it can simulate seemingly random occurrences in communication systems - like network delays and node failures. Apart from probability distributions, probabilistic automata are also used to generate these random occurrences. They consist of a transition matrix; i.e. a mapping between the set of initial and final states of a process.

An automaton is an automatic machine that executes a predetermined sequence of operations, by following predetermined instructions or responding to the occurrence of events. By this definition, a simple alarm clock can also be classified as an automaton. However, here we deal with computational automata[1] - intangible automatons that reside on a computing machine. Similar to their physical counterparts, computational automata follow a predetermined sequence of operations while responding to certain events. There are three basic types of automata:

1. **Discrete automata** - These are systems that consist of discrete states like *on, off; running, loading*, etc.

2. **Continuous automata** - These are systems that consist of a continuum of states. They are most effective at the cellular level, where each cell does not have to take discrete state values and can be in-between states.

3. **Hybrid automata** - These consist of systems where digital computational processes interact with analog physical processes. For instance, a room heater turning *on* or *off* (digital computational process) based on the room's temperature (analog physical process).

Stochastic automata are a generalization of non-deterministic finite state automata[2] - i.e. automata with a non-deterministic state transition function and a finite number of states. What sets the two apart is that stochastic automata come

[1]Automata is the traditional word used for more than one automaton. Automatons is more common today, however, the former is preferred for formal usage.

[2]Stochastic automata and probabilistic automata can be used interchangeably as seen in [36].

with the probability of a given state transition occurring and have their initial state replaced by a vector representing the probabilities of the automaton being in each possible state. Since stochastic automata are at the highest level of generalization, Markov processes and everything discussed in the remainder of this section is a type of stochastic automaton.

There are two basic classifications of stochastic processes. These are based on the following parameters of a process:

1. **Time parameter** - Stochastic processes can either run in discrete or continuous time. This means that events or state changes may occur after distinct time intervals or in continuous time intervals. A continuous parameter implies that it can take any value in a given range. A stochastic process which progresses according to a digital clock could be said to have a discrete time parameter, while a Poisson process is an example of a process with a continuous time parameter.

2. **State space** - Stochastic processes can either work in discrete or continuous state spaces. A discrete automaton (i.e. having states like on/off) is an example of a stochastic process with discrete state space and a cellular level automaton is an example for a stochastic process with continuous state space.

This leads to the Markovian property; whereby, the conditional probability distribution of any future state of a process, depends only on its present state and not on any of the past states, or the amount of time it has spent in the current state. Any process with the Markov property is called a Markov process. Additionally, any process with the Markov property that works in either discrete state space or discrete time is called a Markov chain. A Markov chain having a discrete time parameter can have continuous or discrete states and one with discrete states can have a continuous or discrete time parameter.

Following from the above definitions, we can list out 4 types of Markov processes:

1. **Continuous-time discrete** (state space) **Markov processes** - also called continuous time Markov chains (CTMC).

2. **Discrete-time discrete** (state space) **Markov processes** - also called discrete time Markov chains (DTMC).

3. **Continuous-time continuous Markov processes** - also called jump processes, and

4. **Discrete-time continuous Markov processes** which are also classified as DTMC.

Markov decision processes (MDP) are discrete-time stochastic control processes[3]. Given a set of states, actions, a transition function and a reward function, the process must choose actions in a way that maximizes the reward. However, the process will select the next state based on the transition function (which has inbuilt randomness) and calculate the reward accordingly. If you remove the actions and reward, MDPs reduce to discrete-time Markov chains. Finally, probabalistic timed automata are processes that contain a clock and time-progress conditions. For instance, if the clock reaches a particular value, a decision depending on the current state is made.

Stochastic modelling tools are generally used for the following use-cases:

- **Performance evaluation** - to investigate and optimize the amount of useful work being accomplished.

- **Dependability evaluation** - to assess service continuity using measures like reliability, availability, etc.

- **Formal verification** - To prove that the service delivered (like consensus) satisfies a formal specification of its behaviour.

For the first two use cases, Markov chains are most prominent, since they can represent the temporal dynamics of a system well. For formal verification, labelled transition systems (LTS) are preferred. In LTS, a state change from S to S' implies the occurrence of an action A, which is also the label for that transition. In case of multi-transition states, the choice is usually non-deterministic.

[3]A control process refers to the optimal control theory wherein a process must strive to maximize a given objective function.

This research uses stochastic hybrid automata (SHA) - generalizations that cover MDPs, probabilistic timed automata and labelled transition systems. Since the focus is on performance evaluation, the experiments will use Markov chains (CTMC) to represent the consensus protocols. The tool used in this research is the same as the one used in [12] - a modelling tool called pyCATSHOO [37], which can be used to build Markov chains and conduct statistical analyses on the models, including Monte Carlo simulations. As discussed in Chapter 3; packet loss, network delay and malicious attacks will be simulated in the protocol models. Below, details of the modelling experiments are discussed.

4.2.1 Modelling Consensus

A model was built for each of the four selected protocols - PBFT, Tendermint, Clique and Raft. Each model follows the procedure described by its corresponding protocol, and the model is built to resemble the algorithm as closely as possible. This process is concerned mostly with the consensus and network layers of a blockchain system, and not as much with the execution and application layers. For the network layer, the models adopt a component-view of the blockchain network, consisting of 3 components - a leader, peers, and clients. Each component is represented by a Python class.

Components can be connected using message channels, provided the reference variable being imported, is exported by the component where the variable is defined. Components can communicate and share information with eachother in this way. For instance, the peer can share internal variables like voting information and timeout information with the leader. Since pyCATSHOO does not allow communication between instances of a class, a utility component called the counter is used to count peer votes. The counter collects votes from the peers, counts them and returns a decision to any component that requires it.

The components themselves, contain one or several automatons each with an initial state and a state space. For example, the peer component contains two automatons - a functional and a type automaton. The state space of the type automaton is

- $\{Benign, Malicious\}$ - where Benign is the initial state. Similarly, the functional automaton has the following state space - $\{Start, Waiting, Propose, Prevote, Precommit\}$ - with Start as the initial state. Class instances move from one state to another when events are triggered. For example, the peer may move from the Propose to the Prevote state, when a two-thirds voting majority is reached. Additionally, non-determinism in state transitions can be added by blocking the transition based on a probability distribution, even if the relevant event has occurred. Therefore, the peer will change its state based on the two-thirds majority condition, as well as an exponential probability distribution. The stochastic elements and probability distributions are discussed in more detail later.

Finally, sensitive methods can be called when an instance moves into or out of a state. This is useful to perform state-dependent actions, like broadcasting a block when the leader is in the Ready state. Sensitive methods can also be called when a reference is updated. For instance, when a "majority reached" reference turns true, the relevant procedures may be carried out to change the peer's state. The various state transition diagrams for each protocol are provided in Appendix D.

4.2.2 Assumptions And Liberties

The models discussed above are clearly not the same as full-fledged blockchain applications. This section highlights some of the differences between the two, and discusses the liberties taken while building the stochastic models.

One obvious difference between the models and the applications is that each peer in the models, is just an instance of the peer class. Unlike real blockchain systems, these instances do not have their own unique resources. Another difference is that the message passing is almost instantaneous in the models, which is not representative of real network conditions. However, two bigger differences are the absence of a blockchain data structure in the models, and the use of the counter component.

In the models, when a transaction is committed, important information like the block number and block ID are updated and saved, but the entire block is not saved.

Depending on the use-case, this may not be the best approach, however, metrics like throughput, latency, and success rate can easily be measured without the blockchain data structure. As discussed already, due to limitations with the modelling tool, a counter class is used to tally votes and send the result to other components. In real systems, the tallying occurs at the peers and each peer makes a decision individually. Finally, blockchain platforms like Cosmos and Fabric use small, fixed block sizes, either in terms of memory or transactions per block. This is so that memory requirements of the blockchain are manageable. However, given that an entire block is not saved, the models are not limited by size requirements. Therefore to improve model performance the block size was increased.

Apart from these differences, there exists one fundamental difference between stochastic models and blockchain applications. This is that the former is a model of consensus protocols while the latter are larger applications where consensus is just one component. This means that blockchain applications consist of several components working together, and consensus is just one of these components. For instance, applications provide REST APIs and webpages for users to interact with the blockchain. The models ignore all of these things, and focus only on the consensus algorithm. Below is a summary of protocol-specific decisions taken while building the models.

1. In PBFT, the peers don't work on multiple blocks simultaneously like they do in the real protocol. To do this, each peer instance would need to be in multiple states at once (different state for each block), and pyCATSHOO does not allow for this behaviour.

2. In the Tendermint protocol, there are four steps to reach consensus - propose, prevote, precommit and commit. The first three occur amongst the peers and the commit message is sent from Tendermint to the application layer (i.e. Cosmos). Since our models are concerned only with the consensus layer, the commit message is ignored. Another noteworthy difference in Tendermint is that Tendermint Core implements functionality whereby validators sync their local blockchain copies periodically. This feature is useful in preventing forks, how-

ever it is not implemented in the models.

3. In the models, Clique's timeout period of approximately 150 ms is reduced to a few milliseconds to keep the simulation running[4]. Additionally, a hash puzzle is not calculated while sealing new blocks, instead probabilistic delay is used to replace this. For this protocol's model, it is assumed that only the leader proposes a block in each round. This reduces the overall complexity of the model by avoiding multiple block proposers per round. Finally, the actual epoch threshold for Clique is 30,000 blocks but this is changed to a more manageable number in the model.

4. In Raft, the leader has a heartbeat timeout of around 150ms. Similarly to the model for Clique, these timeout values are changed to a few milliseconds to ensure that the simulations run as expected.

4.2.3 The Stochastic Elements

To emulate real network conditions, stochastic elements like communication delay and byzantine nodes are simulated in the models. To simulate network delay and the time spent in message passing, some delay must be added alongside the relevant state transitions. As discussed earlier, this is done by adding a probability distribution to the state transition. For the experiments, the exponential probability distribution is used, which is defined as,

$$f(x, \lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

[4]The models run for a given number of timesteps rather than a given amount of time. While no blocks are being generated and the functional automaton is idle, the simulations run much faster and sometimes end in a few minutes.

where λ is the rate parameter. All state transitions that depend on voting or sending messages include this probabilistic delay. For instance before the leader sends a heartbeat message in the Raft protocol, an exponential probability distribution is used while entering the state. Similarly peers in PBFT and Tendermint, having received the required majority of votes, must wait for a random amount of time before moving into the next state. In the absence of real P2P communication, this behaviour mimics network delays. As discussed, in the Clique protocol, probability distributions are used before the signer can seal the block, to account for the hash puzzle.

Each model also contains a user-defined byzantine rate, which defines what fraction of the network is byzantine. For instance, a byzantine rate of 2 would result in half the peers being byzantine. A byzantine leader proposes a block with a valid ID to some peers and an invalid ID to others. Thus it sends contradictory information on the network. A byzantine peer on the other hand, will cast contradictory votes on any block that it receives. In the Raft protocol, peers send contradictory messages while voting for the new leader. The only voting in Clique is to vote a malicious node out of the clique; however, adding byzantine behaviour here does not affect the performance of the protocol much, so peers in Clique exhibit no byzantine behaviour.

4.3 Building Blockchain Applications

Building a blockchain application consists of building a communication network amongst the peers, and running a specific application on each machine. The application complexity can vary, but building the network is typically common. Usually, a small set of commands is used specifically to interact with the blockchain; in Ethereum, these commands are written in smart contracts and in Fabric they are written in the chaincode. They are used by other components of the application or SDK to interact with the blockchain ledger. The sections below describe the process of creating the blockchain network and using the API to build applications.

4.3.1 The Network

A network is a collection of connected nodes that communicate with eachother and can be built virtually on a single machine or across several individual machines. Each node is bound to a port of the machine/s and communication between those ports is enabled[5]. One way to build the network is using Docker [38]. A docker compose file defines the network configuration for each node i.e. its address, which ports it listens on, its environment variables and its data volumes. Once connected, the nodes can communicate in accordance with the selected consensus protocol. Another way to build a virtual network is by creating several virtual machines and connecting them together.

However, before building the network, it is important to generate the genesis block and the users' keys. The genesis block encodes important information about the network like the list of validators, their addresses, the block period, consensus algorithm to be used, etc. It is the first block of the blockchain and is common for all nodes on the network. The user key is generated using tools provided by each platform, and is used by accounts to sign their transactions. For the applications built here, each validator is an account holder, however, this need not be the case always.

Amongst the platforms used in this research, the networks on Sawtooth (PBFT), Cosmos (Tendermint), and Fabric (Raft) were built using docker. The Geth (Clique) network was built using virtual machines. Sawtooth and Fabric allow parameters (like addresses and ports) to be specified for each docker container, whereas Cosmos does not. Since Cosmos works only in the application layer, it does not provide tools to configure the network components. The network is created automatically by Cosmos with the help of Tendermint Core. However, if an application is built using Tendermint Core instead, network parameters can be configured using a docker compose file.

[5]Processes are bound to ports; nodes may run one or more processes.

Hyperledger Sawtooth

Each node on the Sawtooth network is made up of four components - the REST API endpoint, a consensus engine, a validator and an intkey transaction processor. One docker container is created for each component and the relevant ports are connected to enable communication amongst the nodes. The intkey transaction processor can process transactions in the form of a key-value pair. For each node, the REST API is exposed and used to interact with the Sawtooth network.

Geth

On Geth, the geth libraries are used to generate the genesis block and start the blockchain network from the command line. Geth also allows the use of JSON-based remote procedure calls (RPC) to build a network using various programming languages. The virtual machines were created using the multipass Linux package. The following steps were followed to build the Geth network:

- Create validator accounts (address, password, keys) on different virtual machines.
- Create the genesis block with Clique consensus, the designated block creators, and account balances.
- Compile each node's address into a static node list, which is shared amongst the validators.
- Start all the nodes using the geth command.

Hyperledger Fabric

The Fabric network is a little different from the others. It consists of organizations which are in turn composed of peers including anchor peers which receive communication from other organizations, endorsers which endorse individual transactions and committing peers which store the blockchain data. The organizations are connected

to eachother via channels and each channel has its own shared ledger as well as chaincode. The orderer nodes may belong to a single orderer organization, or may be split up amongst the other organizations. They order the transactions using the consensus protocol and broadcast their decision to each organization. Fabric also uses certificate authorities for each organization, which generate communication certificates for the member peers. Therefore, if peer-1 of Org-A sends a transaction to peer-2 of Org-B, peer-2 will check that the certificate has been generated by Org A's certificate authority and is valid. All of these network components are created using docker.

4.3.2 The Application

Once the network is ready, an application can be deployed over the network. The application consists of business logic, external databases, wallets, and other components based on the platform. Of these, business logic is the central component of the application. It defines how users and other application components interact with the blockchain. For instance, in this research the business logic for each application is summed up as:

- User A sends funds worth x units to User B.
- User A's account balance is decreased by x units.
- User B's account balance is increased by x units.

This is the logic for a simple asset transfer application, however, depending on the use case, an application can add more functionality like user registration and account creation. In general, there are two aspects to the application logic - handling client data and interacting with the blockchain. The easiest way to interact with the blockchain is using an API or RPC, whereas handling client data entails accepting user input and compiling it in a way that will be accepted by the API or RPC.

Hyperledger Sawtooth

Sawtooth provides SDK libraries for popular programming languages, using which, transactions can be created, batched together and signed. They are then sent to any validator node for processing. The intkey transaction processor on Sawtooth is the default transaction processor. It allows the creation of new accounts with a starting balance, modification of an account's balance, and listing the balance of one or more accounts. This transaction processor works perfectly for our application, however, for more complex applications a custom transaction processor can be created.

Cosmos

Cosmos lets developers create their business logic using modules, and provides an API to interact with Tendermint Core through the ABCI. Cosmos also provides a scaffolding tool called Starport, which can be used to build a template application, rebuild an application, or run it from its last state. The template is built with 9 modules. Of these 8 are pre-built modules and 1 is a custom module which can be modified to fit any business logic. Each of the 8 pre-built modules interacts with the blockchain in a different way - for instance, the accounts module deals with user accounts on the blockchain, and the banking module deals with transacting using the application token. More complex applications can make use of pre-built modules like the slashing and staking modules.

Each module has its own handler and keeper. To modify account balances our custom module needs access to the banking module's keeper. This is done by adding an interface to the banking module's keeper in the custom module. The Starport template also contains a command line utility and a basic web application. This means that users can interact with the application from the command line or using a web page, both of which need to be configured according to the application's business logic. Accounts and validators along with their account balances and stakes, are created at runtime using a config file. Finally, Cosmos exposes three ports for the application - one each for the Tendermint consensus engine, the REST API, and the

application front-end written in Vue.

Geth

Like Sawtooth, Geth also provides libraries for several programming languages and other Ethereum clients like Metamask or Mist. Metamask and Mist are web applications that allow developers to send ether from one account to another. Apart from these libraries, one can interact with the Ethereum blockchain using simple JSON-based HTTP requests. The HTTP request method was used in our experiments. Each node in the network listens for HTTP requests and transaction requests from any user can be posted to any of the node IP addresses.

Hyperledger Fabric

Fabric provides a set of binaries to help with core functionalities. Examples of the binaries include, the fabric certificate authority and the peer binaries, which allow developers to run commands ('fabric-ca' and 'peer' respectively) in the terminal to generate keys, join channels, or interact with a peer. The rest of the business logic is built using bash scripts and common programming languages, therefore, a large part of working with Fabric is working with scripts in the terminal. The chaincode is written in the Go programming language and contains all the functions that enable our application to interact with the ledger. The application itself is written in Node.js and defines the business logic. It also handles key management using wallets and the creation of user accounts. The application interacts with the ledger using the functions defined in the chaincode.

4.3.3 Qualitative Analysis Of Blockchain Platforms

As discussed, each blockchain platform provides different tools to build a blockchain application. The architecture of each platform is also different. This section provides a qualitative analysis for each platform which covers aspects like their capabilities

and specialities, the quality of their documentation, their community, github activity and the associated learning curves.

Platform Abilities

All platforms are extremely modular. Hyperledger Sawtooth and Fabric support several programming languages, whereas the Cosmos SDK and Geth support only the Go programming language. Cosmos SDK allows for inter blockchain communication, provided the blockchains in question use a PoS protocol built with Tendermint core. One of its strongest selling points is that blockchains built using Cosmos do not fork. Sawtooth offers dynamic consensus, which is not offered by any of the other platforms. Hyperledger Fabric is different from the others in that it allows users to build a distributed ledger application and not a blockchain. It also uses certificate authorities and certificates for each organization. Geth allows users to interact with a number of Ethereum networks, including the main net and various test nets. Finally, like Fabric, Geth works best with trusted validators, whereas Sawtooth and Cosmos are expected to work with byzantine validators.

Platform Community And Docs

The Hyperledger Foundation assigns a phase to each of its projects based on where the project is in its lifecycle. These phases are: proposal, incubation, active, promoted release, deprecated, and end of life. The direction for each project and which phase it fits into is decided by the Hyperledger technical steering committee (TSC). The Hyperledger Foundation as a whole, is experiencing global growth, with new members joining every few months.

Of the two Hyperledger projects used in this research, Fabric is by far the more popular one. The project has helpful documentation, but is too vast and can be intimidating for beginners. The project is labeled active and will probably remain so for the foreseeable future. The last significant github update at the time of writing was 5-30 days ago. Sawtooth, the other Hyperledger project, is also labelled as active.

However, it has a mostly inactive community with the last significant github update 7-9 months ago. The documentation is helpful and the guides are easy to follow.

Geth has an active and established community - it is one of the most popular implementations of Ethereum and has useful documentation. The last significant github update for the geth project was 5-30 days ago, and the community closes issues regularly. Cosmos is also an active and growing community. However, it is still young, and the project is still gaining traction. The documentation is detailed and easy to get around. The last significant github update at the time of writing was 5-30 days ago and the community closes issues regularly.

Ease Of Use

Here, it is assumed that a user has some prior knowledge of blockchain concepts. Cosmos is very simple to setup and get started with. There aren't many dependencies and using starport, a working Cosmos application can be ready in less than 15 minutes. However, understanding the code generated by starport, and updating the business logic may take longer. Fabric on the other hand, requires a slight learning curve from the get go. Getting used to the Fabric architecture goes a long way in getting used to the platform. One advantage of working with Fabric is that for any issue, it is likely a solution can be found on the internet. The same cannot be said for Cosmos. Unlike Cosmos and Fabric, Sawtooth and Geth are relatively easy to get around. Using Sawtooth's intkey transaction processor, or Geth's web3 libraries, is simpler compared to building applications in Fabric or Cosmos, but they serve the same purpose. For production ready and more complex applications, Sawtooth and Geth also allow more configurable applications to be built using a different transaction processor and different tools in the web3 libraries respectively.

4.4 Tools To Measure Performance

The three performance metrics - average write throughput, average write latency and success rate - were measured for each protocol model and application. In the models, the spread of the peers' chain lengths was found to be more useful than success rate when stochastic elements were introduced. The process of performance measurement is discussed in the following passages.

It is easy to measure the metrics using timers and counters in the models. For more complex metrics, pyCATSHOO allows tracking of model parameters which can be analysed later. For load tolerance, the input transaction workload is varied for the models, and for fault tolerance byzantine activity and delay are introduced. As discussed earlier, a byzantine leader will send proposals with different block IDs to different nodes and a byzantine peer will vote on blocks arbitrarily by sending contradicting votes. With these changes made, the primary metrics are calculated and recorded once again.

Three of the blockchain platforms provide usable HTTP request endpoints. Using Cosmos and Geth, transactions can be created and sent to the network without them appearing on the blockchain. This is usually used to check the result of a transaction without recording it. However, Cosmos provides only this functionality using the HTTP endpoints. The only way to write to the blockchain is through the command line or using gRPC which is not supported in the version of Cosmos used for this research. Therefore, performance measurement of the Cosmos application is done using the command line tool. Three python scripts using the tmux Linux package, are used to open multiple terminal sessions. Each session sends transactions to the Cosmos application simultaneously and the performance metrics are calculated over time using timers and counters. The number of sessions and the number of transactions are configured to generate different amounts of load. The results from each session are compiled to get the final results. Since the Tendermint validators cannot be accessed from Cosmos, the chaos tests for fault tolerance are not performed.

For Sawtooth, Geth and Fabric, a load testing tool called Locust [39], is used to

generate load in terms of the number of concurrent users. Locust swarms requests to the port where the HTTP server is running, and depending on the response received, each request is classified as a success or a failure. The tool automatically calculates the successful requests per second (throughput) and the average response time (latency). To emulate real network conditions a tool called Pumba [40] is used at each node (docker container) to add network loss, delay, corrupted messages to mimic byzantine faults, and paused nodes to mimic crash faults. Pumba uses the iproute2 Linux package under the hood, but since it works only with Docker, and our implementation of Geth does not use a docker network, iproute2 is used directly in the Geth network.

To compare the results obtained from the models and applications, the metrics are plotted over time and the resultant graphs are compared. These results are presented and discussed in the next chapter.

CHAPTER 5

Results

5.1 Experiment Details

The results include comparison charts of the models' performance, baseline performance of the apps, and the change in the baseline during load and chaos tests. The model and application parameters are provided below in Table 3 and Table 4 respectively.

TABLE 3: Modelling simulation parameters

Parameter	**Value**
Number of peers	6
Transactions per block	70
Input workload (tx/sec)	1000, 5000, 10000, 15000
Maximum simulation timesteps	500000
Exponential distribution rate parameter (λ)	2
Byzantine rate	0, 2

In the models, throughput, latency, and standard deviation of local chain lengths (σ), are calculated against a varying workload. For Raft, the overall success rate is calculated instead of σ. The metrics are calculated four times for each protocol model - once with byzantine nodes, once with simulated delay, once with both delay and byzantine nodes, and finally, once with no stochastic elements (i.e. the baseline).

In the applications, throughput, latency, and success rate are calculated at a

TABLE 4: Application parameters

Parameter	Value
Number of validators	6
Block size	10 tx/block *OR* default in MB
Baseline user load	250, 50
Load test user loads	250, 500, 1000, 1500
Locust workers	3
Users per second per worker	1, 2

constant input load. We call this the baseline results. The throughput and latency are also measured while varying the load and while adding faults to the blockchain network. These are called the load and chaos tests respectively. We also summarize the chaos testing results by providing the average value for each metric (throughput and latency) while each network fault is being injected into the network.

5.2 Discussion of Results

5.2.1 Stochastic Models

In Figs. 5-8, performance in the presence of simulated delays or byzantine nodes generally falls in-between the baseline and the case where delay and byzantine activity occur together. Fig. 5 shows the change in throughput for each protocol as the input transaction workload is varied. For PBFT, Tendermint, and Raft, the baseline throughput is well separated from throughput measurements in the presence of stochastic elements. For the Clique protocol model, adding network delays and byzantine nodes does not affect throughput as much as it does in the other models. This is because of Clique's leader-follower architecture. In the other protocols, when half the network is byzantine, consensus cannot be reached due to contradicting votes being sent across the network. Due to this, system throughput degrades. However,

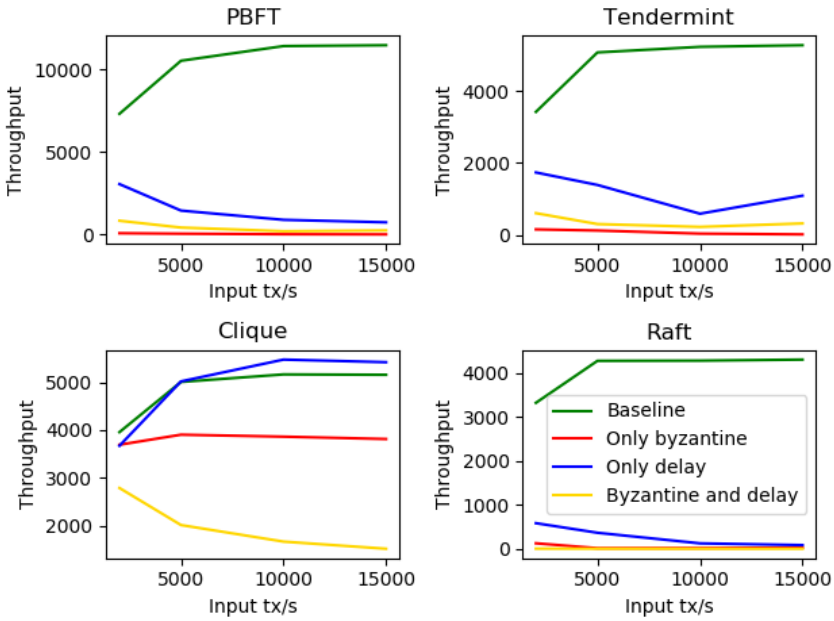

FIGURE 5: Modelling: Write throughput

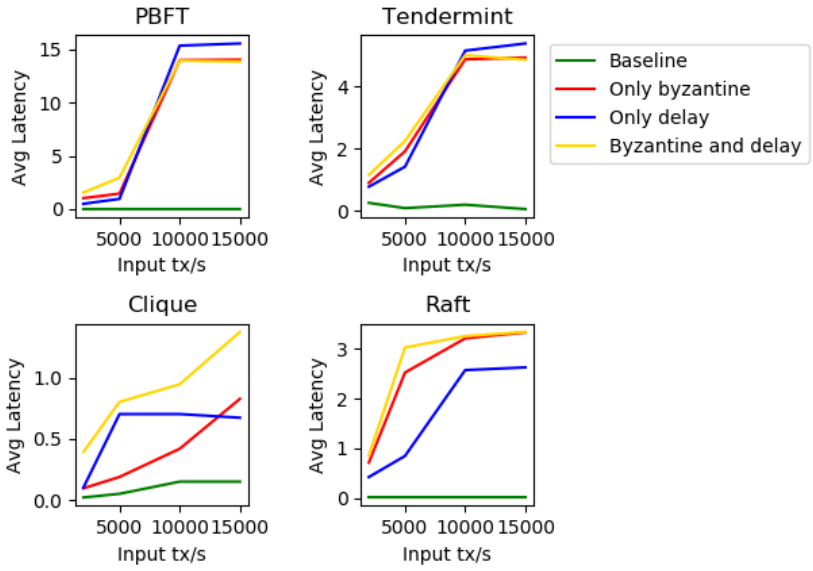

FIGURE 6: Modelling: Average latency

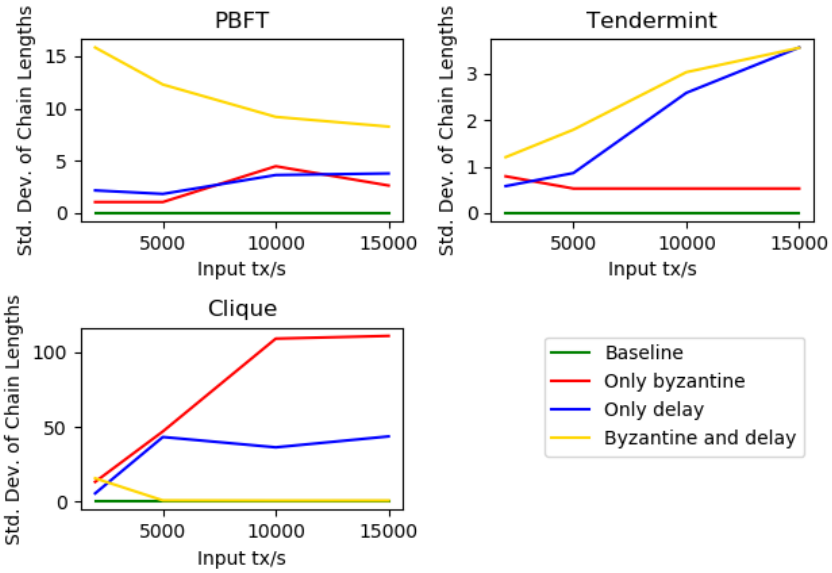

FIGURE 7: Modelling: Standard deviation of chain lengths

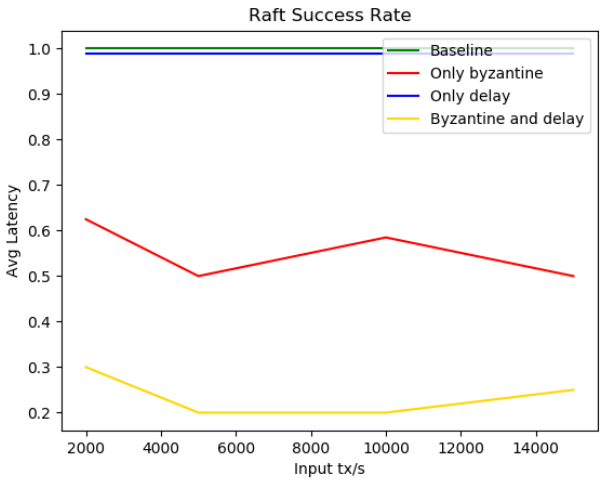

FIGURE 8: Modelling: Success rate of Raft

since the peers in Clique do not communicate before adding a block to their local chains, each peer simply accepts or declines the block proposed by the leader based on validity. If the leader itself is byzantine, this may lead to inconsistent local chains, but since a portion of the network receives a valid proposal, the overall throughput of the system does not degrade. Overall, PBFT shows the best baseline performance in terms of throughput, while the other three models post comparable results. The addition of byzantine nodes generally affects throughput more than network delays do.

Fig. 6 shows the change in average latency for each protocol as the input transaction workload is varied. The results for the PBFT and Tendermint models are almost identical, although Tendermint has lower latency. As with throughput, latency results in the presence of stochastic elements are similar and well separated from the baseline. The similarity in PBFT and Tendermint latency results is understandable since both protocols use the same BFT-based voting rounds, although Tendermint uses one more round of voting than PBFT. The latency results for Raft also follow a similar pattern to the ones for PBFT and Tendermint, however there is some separation amongst results in the presence of stochastic elements. For the Clique protocol model, although the results are well separated, in terms of magnitude there is not much difference in results no matter what stochastic elements are added. Interestingly, the average latency for Clique in the presence of byzantine nodes, does not flatten out like it does with the other models.

When simulations were run with byzantine nodes and delay, the success rate for PBFT, Tendermint, and Clique models was different for each local blockchain copy. This is because the nodes receive either contradicting or delayed messages, which results in different nodes reaching different conclusions at the end of each round (no consensus). In other words, some nodes might add a block to their local blockchain, while others might not. For this reason, the standard deviation of local chain lengths is used to quantify this inconsistency in lengths amongst the local chains. From Fig. 7 it can be seen that for PBFT and Tendermint, adding byzantine failures with network delay caused the local blockchains to diverge the most. For Clique, adding both faults

together did not affect the local chains as much. Overall, the addition of stochastic elements affected the crash fault tolerant Clique model more than it did the byzantine fault tolerant models. The spread of chain lengths in the Tendermint model is the smallest, while it is slightly larger for PBFT and considerably larger for Clique.

Since delay and byzantine nodes did not affect the consistency of local chains for the Raft model, it is left out from Fig. 7. This means that the local chains in the Raft protocol did not diverge during simulation. However, the presence of byzantine nodes did have an effect on Raft's overall success rate as shown in Fig. 8. Since all local chains are consistent, the success rate at each node is identical and is called the overall success rate. In Fig. 8, the baseline success rate for Raft is equal to its success rate when network delays are simulated, they are separated in the plot for visibility.

5.2.2 Blockchain Applications

The load generated for the blockchain applications is the total number of users interacting with the application, as opposed to number of input transactions per second for the models. In Table 5, a manageable load was selected for each protocol in order to get as stable results as possible. For PBFT (Hyperledger Sawtooth), Tendermint (Cosmos), and Clique (Ethereum), 250 users were manageable. However, Raft (Hyperledger Fabric) could not deal with the same load of 250 users. This is down to how endorsement works in Hyperledger Fabric rather than due to the protocol itself.

TABLE 5: Application baseline performance

Protocol	**Write Throughput (tx/s)**	**Avg. Latency (ms)**	**Success Rate**	**User Count (Load)**
PBFT	50	1100	0.88	250
Tendermint	93.1	2039	1.0	250
Clique	27.3	49	1.0	250
Raft	5.8	1850	0.98	50

When a peer validates a transaction in order to give its endorsement, it processes the transaction and obtains the resultant ledger state, called the read set. After the transaction is accepted and ordered, while being committed, it is processed once

again and the resultant ledger state is called the write set. If the read and write sets do not match the transaction is cancelled. This is not ideal for applications expecting large workloads (or our load test) because the state changes several times between endorsement and committing of a single transaction. Companies like Boxer Construction Analysts and Robinson Credit Company have implemented independent solutions to deal with this issue [41].

One final note on Tendermint explains why its average latency is so high compared to the other applications. Each account registered on the Tendermint network has an account number and a sequence number. The sequence number is incremented by the app every time the account makes a transaction. However, the internal copy of this sequence number only changes once the blockchain state is updated. While processing new transactions, the sequence number of the sending account is checked against its internal copy. If the two values do not match, the transaction is cancelled. In other words, the application cannot accept new transactions from a given account, until the account's last transaction has been accepted (committed). Given the large number of concurrent users, each one ends up waiting for older transactions to be committed, which affects the average latency of the application. Overall, Tendermint seems to be the best in terms of throughput, and Clique in terms of average latency. However, Raft may perform better if Hyperledger Fabric is configured to deal with higher loads.

The load tests for each application were carried out until the application crashed, or performance degraded visibly. PBFT (Fig. 9) did well till the load reached 1000 users, after which it quickly degraded. PBFT's throughput fluctuates when the load is changing, but stabilises once the load stabilises. Similarly, average latency degrades when the load is increasing but stabilises when the load stabilises. Tendermint (Fig. 10) and Clique (Fig. 11) showed the best performance under load. Both started degrading when they hit 1500 concurrent users. The Geth application crashed once it reached 1500 users which is what caused its performance to degrade. As discussed, Raft (Fig. 12) performed the worst under load. Hyperledger Fabric's inability to naturally handle large loads explains why the performance is stable at lower loads but

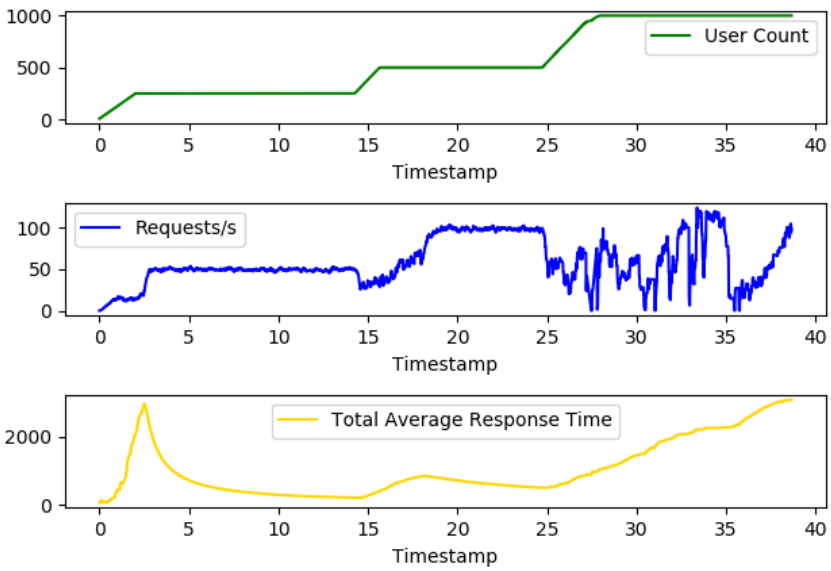

FIGURE 9: Load test: PBFT

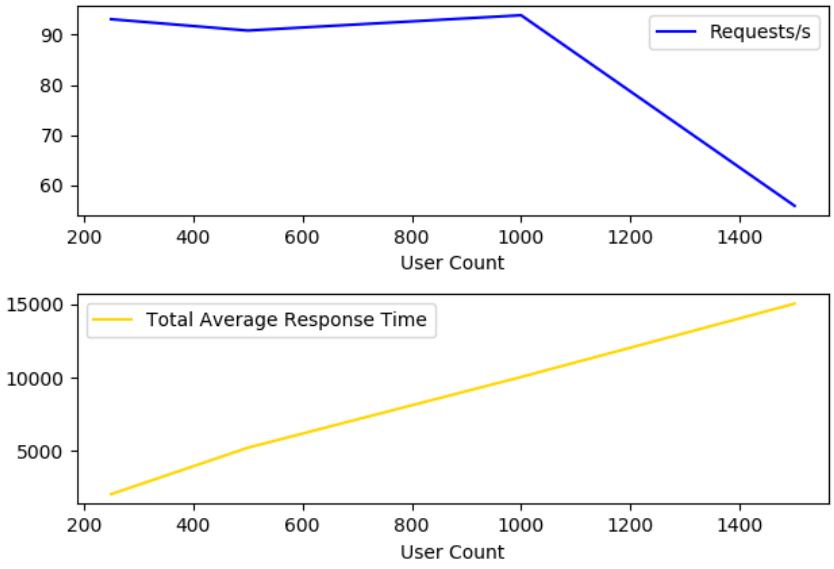

FIGURE 10: Load test: Tendermint

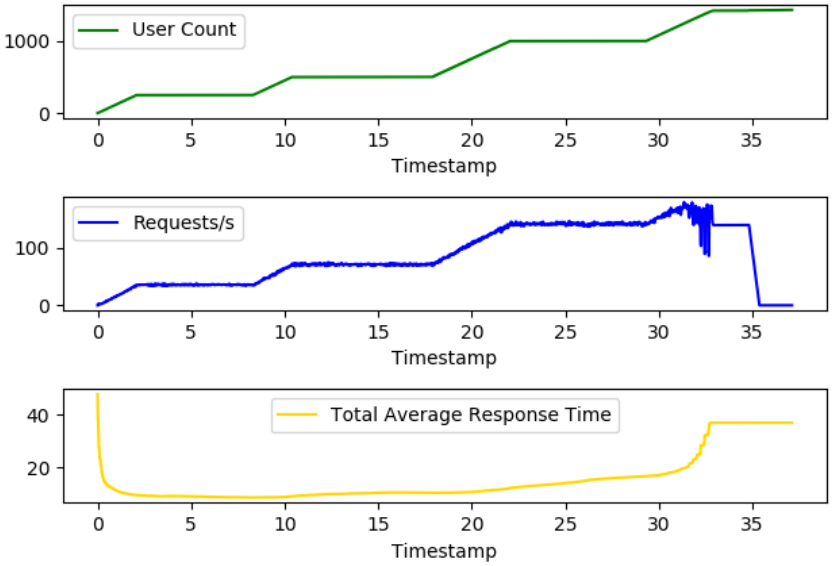

FIGURE 11: Load test: Clique

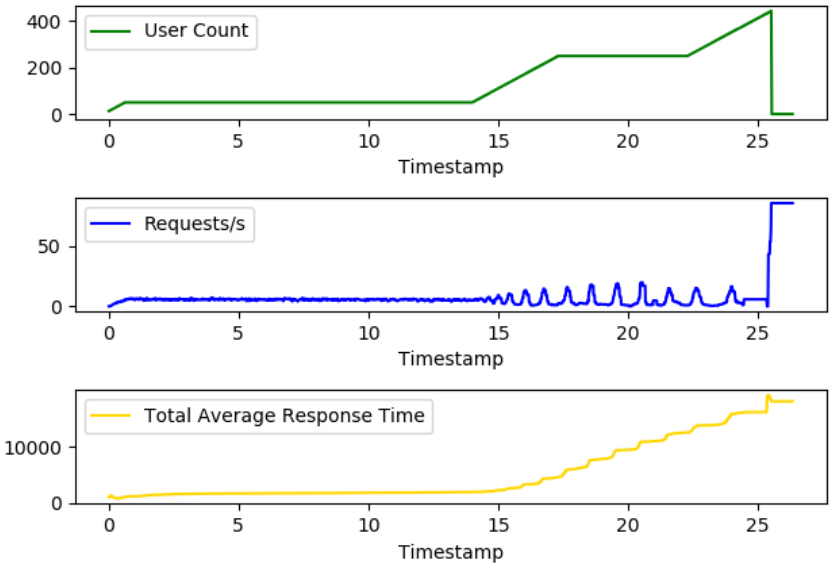

FIGURE 12: Load test: Raft

starts degrading / oscillating before even 250 users are spawned.

TABLE 6: Average of performance metrics under varying network conditions

Protocol	Metric	baseline	delay (100ms)	loss (15%)	delay+loss	corrupted (50%)	corrupted+delay+loss	paused (half)
PBFT	Throughput(tx/s)	50	17.5	16.2	24.78	10.5	16.5	4.9
	Median Latency(ms)	18	4463	20.88	4475	2055	4513	Null
Clique	Throughput(tx/s)	27.3	28	28.5	28.5	25.76	24	5
	Median Latency (ms)	6	105	6	110	7	103	Null
Raft	Throughput (tx/s)	5.8	5	4.8	3.75	3.82	3.55	2.33
	Median Latency (ms)	1766	3150	3300	5100	6271	6430	18500

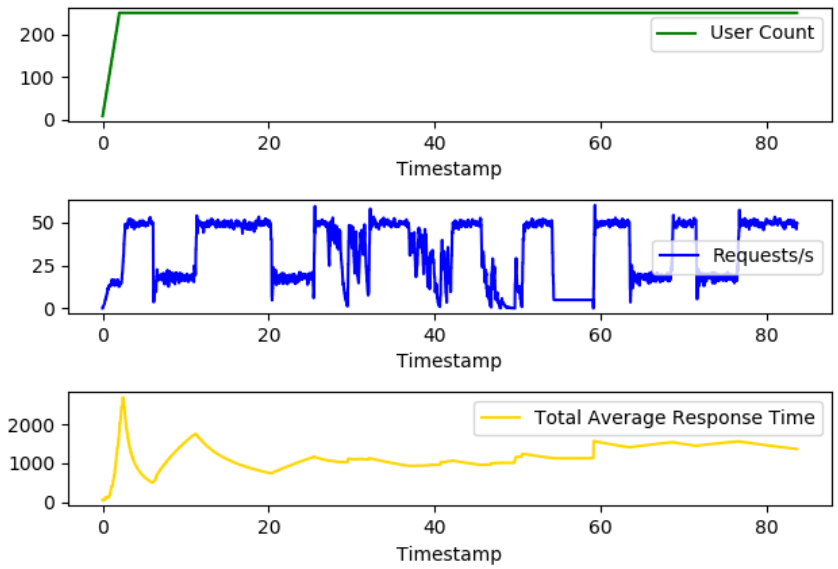

FIGURE 13: Fault tolerance: PBFT

The chaos tests for each application were conducted at the same constant load as their respective baseline tests. The faults introduced during the test were (in order): delay, loss, delay and loss, corrupted messages from a single node, corrupted messages from half the network, corrupted messages (1 node) with delay and loss, corrupted messages (half network) with delay and loss, paused nodes. Here, corrupting outbound messages has a similar effect to byzantine activity, since each node receives contradicting messages. Similarly, pausing nodes is similar to crash failures.

The metric values when certain network faults were injected, are specified in Table 6. Figs. 13-15 depict the entire test during which the faults were simulated

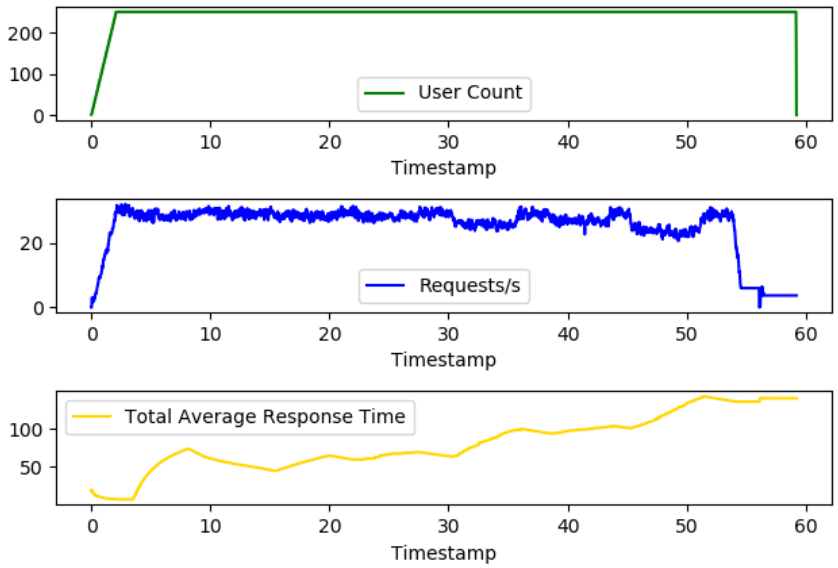

FIGURE 14: Fault tolerance: Clique

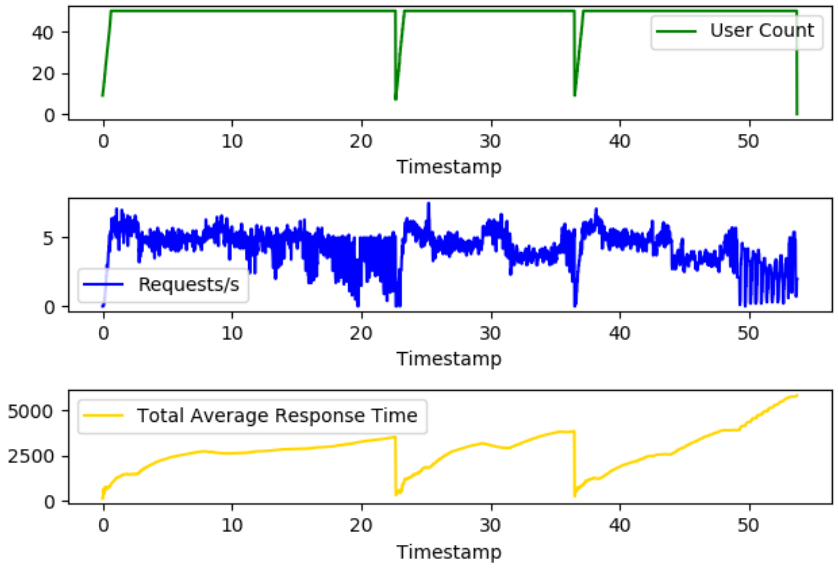

FIGURE 15: Fault tolerance: Raft

consecutively. In these test runs, following each fault mentioned above, the network was returned to normal conditions for an equal period of time, before injecting the next fault. This can be observed in Fig. 13 where throughput returns to the baseline periodically. The throughput in these plots can be compared to the throughput in Table 6. However, the latency in Table 6 refers to the median latency at each instant during the test, while the latency in Figs. 9-15 represents a running average of the latency throughout the entire test run.

Entries with a 'Null' value in Table 6 signify that there is no data available for that period of the test. This is usually accompanied by a few short spikes where the latency metric degrades heavily. While the median response time (latency) may remain relatively low during the spike, the maximum response time shoots up. For instance, when half the network was paused, the maximum response time degraded to 300000 ms in PBFT and 28000 ms in Clique. Apart from these short spikes, there is no data for latency during the periods in question. Pausing half the network nodes has the most drastic effect on performance compared to the other faults simulated. One noteworthy observation is that network faults affect the throughput of PBFT drastically, but have very little effect on Clique's throughput. On the other hand, the average latency of Clique and PBFT does not change drastically, whereas Raft's average latency is continuously degrading as different network faults are added and removed from the network.

The Fabric application could not handle the test very well and crashed thrice, hence the drops in the plots of Fig. 15. In fact, this figure consists of three separate tests whose results were combined together. The throughput plot for Raft looks like it fluctuates a lot, but this is due to the scale of the y-axis and in reality the extrema are not separated by much in the absence of network faults. Similar to Clique, the faults affect Raft's latency more than its throughput. It can also be seen that Raft handles network delay or loss well compared to other faults.

5.3 Final Thoughts

Although the application results in Table 6 and the stochastic model results in Figs. 5 and 6 are not comparable in terms of magnitude, they follow the same overall trends. For example, the throughput of PBFT is best with only delay, followed by corrupted messages with delay, and then corrupted messages without delay. Fig. 5 gives the same relative order. The stochastic models show the baseline throughput performance follows the order: PBFT, Clique, Tendermint, and Raft from best to worst. The blockchain application results follow the order: Tendermint, PBFT, Clique, and Raft. This shows that the Tendermint application does much better and the PBFT application much worse than the models predicted. Similarly for average latency, the models predicted the following order from best to worst: Clique, Raft, Tendermint, PBFT; while the applications showed the following order: Clique, PBFT, Raft, Tendermint. Here, PBFT does much better than the models predicted, while the others performed as expected.

It is important to also consider the role of platform architecture in these results. As discussed, Hyperledger Fabric (Raft) and Tendermint (Tendermint) follow certain rules that have an adverse effect on application performance. Since the models did not take into account the account sequence numbers, they could not have predicted the degradation in Tendermint's latency. Similarly, if the Fabric application was built to handle a larger load as in [41], it would definitely improve the application's latency results. However, this needs to be verified. Overall, the models give a good understanding of how different protocols handle load and network faults. They also give a decent overview of the protocols' relative performance, however, it must be kept in mind that in addition to consensus protocols, blockchain platforms play an important role in the performance of blockchain applications as well.

CHAPTER 6

Conclusion

This research conducted a comparative analysis of four permissioned blockchain platforms using blockchain deployments and stochastic consensus protocol models. It studied the use of stochastic modelling in measuring system performance, and compared our modelling results against results obtained from blockchain applications. We also studied the effect of various network faults and input workloads on model and application performance. The results followed a similar trend in both models and applications. However, we found that in addition to the consensus protocol used, the architecture of blockchain platforms also plays an important role in determining system performance. Therefore, stochastic protocol models are useful while predicting relative performance of consensus algorithms, but results must be verified using blockchain deployments. Finally, we provided a qualitative analysis of the selected platforms based on their capability, usability and popularity.

In the future, we can conduct performance tests using newer versions of blockchain platforms wherever applicable. For instance, the latest version of Cosmos SDK introduced breaking changes which have a significant effect on our methodology. We can also implement the high-throughput network as described in [41]. The chaos testing scenarios used in this research can be extended to design a complete chaos test suite for blockchain applications. In addition, applications with more intricate business logic could also be tested. Using cloud services to measure the geographic scalability of blockchain applications is another task for the future. Finally, it might be useful to investigate the overhead introduced by different blockchain platforms, by comparing different platforms that offer the same consensus protocol.

APPENDIX A

Definitions

1. **Finalty**: Finalty is a guarantee that past transactions will not change i.e. will be final. Most PoW-based protocols offer probabilistic finalty, which means that transactions will be finalized *eventually* with increasing probability after every new block is added. Others, like some of the PoS-based protocols (including Tendermint), offer immediate finality i.e. once a new block is added it is immediately finalized.

2. **Poisson processes**: A Poisson process is a process wherein consecutive events occur completely randomly, but the average time between the occurrence of any two consecutive events is constant. These processes are said to follow the Poisson distribution. Consider a car-wash with a capacity of one car, where cars arrive randomly one after the other. If the average time between consecutive arrivals (say, over a period of one year) at the car-wash is constant, this process would qualify as a Poisson process.

3. **Directed Acyclic Graph (DAG)**: A DAG is a directed graph that contains no cycles i.e. starting from one vertex and following the directed edges, you can never reach the starting vertex again. Additionally, the edges must flow in a common overall direction.

4. **Monte Carlo Simulation**: Monte Carlo simulations are used to predict the probability of an outcome, given a set of fixed occurrences and probability distributions. This is done by running several simulations and calculating the probability of certain outcomes occurring.

5. **API**: An API is an interface that exists between a program offering some functionality, and an application that wants to use that functionality. For example, the functions that allow an application to add, update, or delete blockchain records, make up an API. A REST API, is a type of API that uses certain rules and works over HTTP.

6. **RPC**: Using an RPC entails running a procedure remotely (i.e. on another system or network), as if it were a local procedure call.

7. **SDK**: An SDK is a collection of tools and methods that usually abstract over a lower-level program. For instance, Cosmos-SDK creates an abstraction over Tendermint Core.

APPENDIX B

Consensus protocol pseudocode

B.1 PBFT

Algorithm B.1: PBFT pseudocode

```
while(no_consensus):
    selectLeader("round_robin")
   # leader actions
   validateTransactions()
   block = createBlock()
   # view= current round number
   broadcastMsg("preprepare", view, block.ID, block)

   # validator actions
   receiveMsg("preprepare", view, block.ID, block)
   validateMsg():
      # Check 1: no other block with same view and block.ID.
      # Check 2: view is the current view.
      # Effect: if block is invalid wait for timeout.
   broadcastMsg("prepare", view, block.ID, validator.ID)

   # if received 2/3 prepare, broadcast commit
   count = 0
   while (count < 2*(tot_peers-1) / 3) and (!timeout):
```

```
      if receiveMsg("prepare"):
         count++
   if (count >= 2*(tot_peers-1) / 3):
      broadcastMsg("commit", view, block.ID, validator.ID)

   # if received 2/3 commit, add to local chain
   count = 0
   while (count < 2*(tot_peers-1) / 3) and (!timeout):
      if receiveMsg("commit"):
         count++
   if (count >= 2*(tot_peers-1) / 3):
      addToChain(block)

# global actions
# change view
if (timeout):
    # block.ID of last globally accepted block
   broadcastMsg("change_view", view+1, block.ID, validator.ID)

# periodically erase logs
if view % 10 == 0:
    # 10 is set arbitrarily
   eraseLogs()
```

B.2 Tendermint

Algorithm B.2: Tendermint pseudocode

```
# block_id generated with timestamp
while(no_consensus):
    # weighted according to stake
   selectProposer(weighted_round_robin)

   # PROPOSE STEP
   # leader actions
   if locked:
      broadcastMsg("propose", round_num, block.height, lock.block,
          block.ID, lock.proof)
   else:
      validateTransactions()
      block = createBlock()
      broadcastMsg("propose", round_num, block.height, block, block.ID)

   # validator actions
   # if received proposal then b/c prevote.
   blocks = []
   receiveMsg("propose", block.ID)
   # blocks is used to check if propose was recd
   blocks.append(block_id)
   if lock.proof:
      if checkProof(lock.proof):
         broadcastMsg("prevote", round_num, block.height, block.ID)
   else:
      broadcastMsg("prevote", round_num, block.height, block.ID)

   # if recd 2/3 prevotes + propose then b/c prevote (if not done yet).
```

```
count = 0
while (count < 2*(tot_peers-1) / 3) and (!timeout):
   if receiveMsg("prevote", block.ID):
      count++
if (count >= 2*(tot_peers-1) / 3) and (block.ID in blocks):
   broadcastMsg("prevote", round_num, block.height, block.ID)

# if recd 2/3 prevotes, lock and b/c precommit
count = 0
while (count < 2*(tot_peers-1) / 3) and (!timeout):
   if receiveMsg("prevote", block.ID):
      count++
if (count >= 2*(tot_peers-1) / 3):
   if (locked) and block.ID > lock.ID:
      lock(block.ID):
         lock.ID = block.ID
         lock.block = block
         lock.proof = generateProof()
   broadcastMsg("precommit", round_num, block.height, block.ID)

# if recd 2/3 precommits, req commit (add block to chain)
count = 0
while (count < 2*(tot_peers-1) / 3) and (!timeout):
   if receiveMsg("precommit", block.ID):
      count++
if (count >= 2*(tot_peers-1) / 3) and (block.ID in blocks):
   addToChain(block)
   block.height += 1
   round_num += 1
   # if current block is locked, unlock it
   if lock.ID == block.ID:
      locked = False
```

```
        lock.ID = None
        lock.block = None
      continue
```

B.3 Clique

Algorithm B.3: Clique pseudocode

```
while(!timeout):
   selectSigner(round_robin)

   setHashDifficulty(2)
   if signer_down:
      wait(signer_num * 500)
      if signer_down:
         setHashDifficulty(1)
         continue

   # signer actions
   voted_against = {}
   validateTransactions()
   calculateHash()
   block = sealBlock()
   # votes if there is ongoing voting else null
   broadcastMsg("addBlock", round_num, epoch, block, block.hash, votes)

   # peer actions
   voted_against = {}
   block = receiveMsg("addBlock")
   validateHash(block.hash)
   if valid:
```

```
      addToChain(block)
      block.count += 1
      round_num += 1
   else:
      continue

# global actions
# voting against a peer
if remove_peer:
    # peer.ID of member to remove, voter.ID of voter
   broadcastMsg("remove_peer", round_num, epoch, peer.ID, voter.ID)

if receiveMsg("remove_peer", peer.ID, voter.ID) and (agree):
   if voter.ID not in voted_against[peer.ID]:
      # total votes are stored for each peer
      voted_against[peer.ID].append(voter.ID)
   # notify network of vote
   broadcastMsg("remove_peer", peer.ID, voter.ID)

for peer.ID in voted_against.keys():
   if len(voted_against[peer.ID]) >= threshold:
      removeUser(peer.ID)

# updating epoch
if (block_count == 30000):
   # add empty block, reset count, settle votes
   addToChain(null)
   block.count = 0
   epoch += 1
   settleVotes():
      calculateVotes(peer.ID)
      removeUser(peer.ID)
```

```
    resetCounts(vote_against)
```

B.4 Raft

Algorithm B.4: Raft pseudocde

```
# Leader Election
if time_since_heartbeat > node_timeout:
   while (!leader):
      ## candidate actions
      setCandidate()
      # term= round number
      term += 1
      castVote(self.ID)
      # log.index= index of last log entry
      # log.term= term of last log entry
      broadcastMsg("request_vote", term, peer.ID, log.index, log.term)

      ## peer actions
      if (!voted) and receiveMsg("req_vote") and (term <= current_term) and
          (candidate.logUpdated):
         castVote(candidate.ID)
         time_since_heartbeat = 0
      else:
         declineVote()
            setCandidate(self.ID)
            # ..same procedure as candidate before

    ## global actions
    if receiveMsg("append_entries"):
      # i.e. received heartbeat
```

```
      break
    calcVotes()
    if majority_exists:
      # peer.ID of the peer with majority votes
      setLeader(peer.ID)
      leader = True
      break
    else:
      leader = False
      continue


# Log Replication
## leader actions
if (time_since_heartbeat==100) and (new_transactions):
   # send new log entries
   validateTransactions()
   updateLog():
      # update with new_log_entry
      # get new log.index
      # get new log.term iff this is the first log update in current term
   broadcastMsg("append_entries", term, leader.ID, log.index, log.term,
       log.entry)
   time_since_heartbeat = 0
elif time_since_heartbeat == 100ms:
   # send heartbeat signal
   broadcastMsg("append_entries", term, leader.ID, log.index, log.term,
       null)
   time_since_heartbeat = 0


## peer actions
```

```
if receiveMsg("append_entries") and (term >= current_term) and
    (peer.logUpdated):
    # peer.logUpdated= peer's logs were up-to-date till now
    # now add new entries
    updateLog()
    replyMsg("received", term, peer.ID, log.index, log.term)

## leader action
# if leader recd >2/3 acknowledgments, commit update
while (count < 2*(tot_peers-1) / 3):
   if receiveMsg("received"):
      count++
if (count >= 2*(tot_peers-1) / 3):
   commitLog()
   broadcastMsg("commit", term, commit.index)

## peer action
if receiveMsg("commit"):
   commitLog()
   # update log.index
   log.index = max(commit.index, log.index)
```

APPENDIX C

Stochastic modelling with pyCATSHOO

The pyCATSHOO framework is written in C++ and offers libraries for Python as well as C++. It uses an object-oriented approach to model piecewise deterministic Markov processes. These are processes whose behaviour follows a mixture of determinism and random state jumps. By using different differential equations for the deterministic component, pyCATSHOO allows different models to be built, including Markov chains and several types of queuing models.

In [12] the authors briefly describe pyCATSHOO's framework as follows. A model is defined as a system of components that communicate through message passing. These components are defined by:

1. The set of variables, $V = I \cup E$, where I and E are internal and external variables of the component.

2. The set of message boxes, B, which declare input and output ports for the component through which external variables are imported and internal variables are exported.

3. The set of automata, A, where an automaton, a, is defined by a 3-tuple, $< S, s_0, T >$ where:

   (a) S = set of all states,

   (b) s_0 = initial state,

(c) T = set of transitions, denoted by $< s, g, d, p >$, where:

i. s = initial state,

ii. g = transition validity (True or False),

iii. d = stochastic delay,

iv. p = probability distribution over the state space to select the final state.

4. The set of evolution rules, $R = C \cup D$, where C and D are continuous dynamics (differential equations) and discrete event rules (functions triggered by certain events) respectively.

pyCATSHOO is used for performance assessment of complex, hybrid systems. It can model both, components of a hybrid system i.e. discrete and stochastic behaviours; and the continuous and physical phenomena that evolve inside a system.

APPENDIX D

Model state transition diagrams

There are two automatons used in the protocol models - one for steps in the consensus algorithms and another to determine whether a node is benign or byzantine. The state transitions diagrams for the former are presented here.

Each model has four automatons - one each for the client, peer, leader and counter. The client state diagrams are the same for every protocol while the others change considerably. The transition between any two states can be summed up by when the transition occurs, and what the result of the transition is. This is specified in the diagrams which begin from the following page.

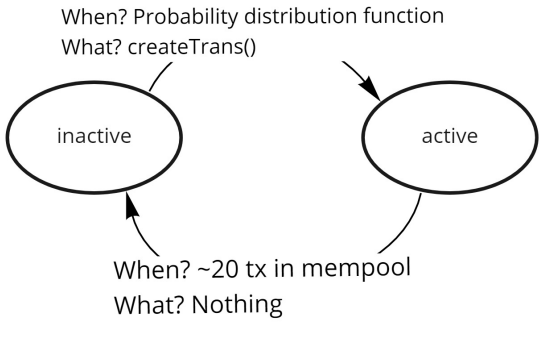

FIGURE 16: State transitions: PBFT Client

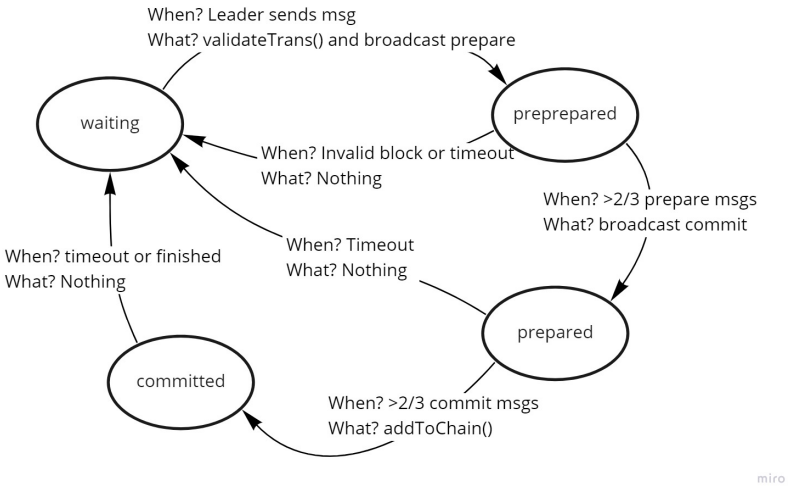

FIGURE 17: State transitions: PBFT Peer

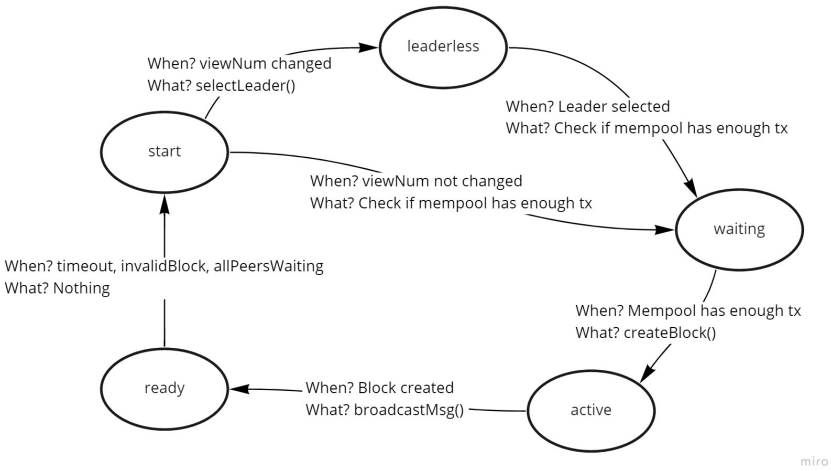

FIGURE 18: State transitions: PBFT Leader

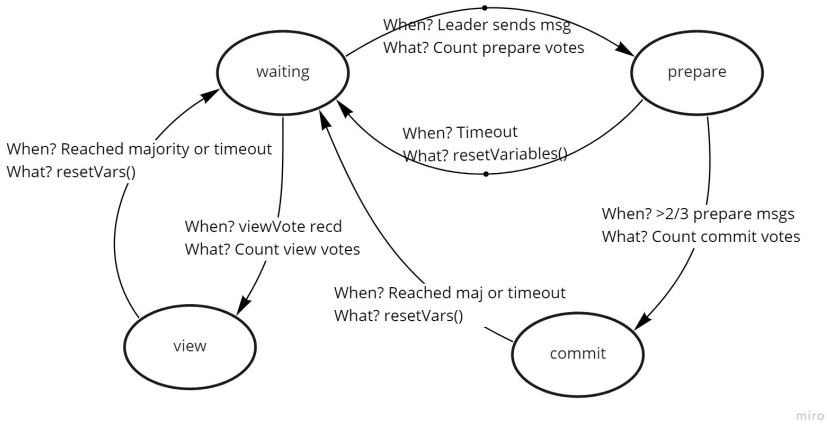

FIGURE 19: State transitions: PBFT Counter

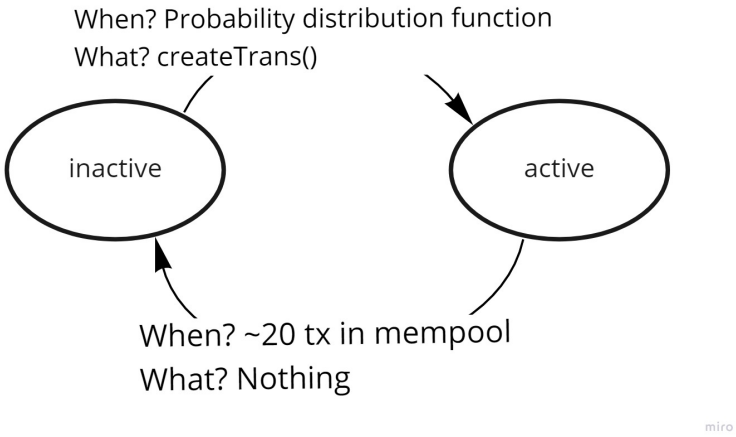

FIGURE 20: State transitions: Tendermint Client

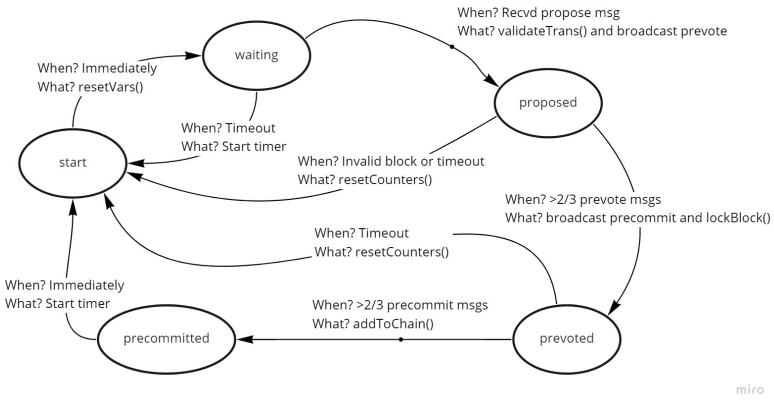

FIGURE 21: State transitions: Tendermint Peer

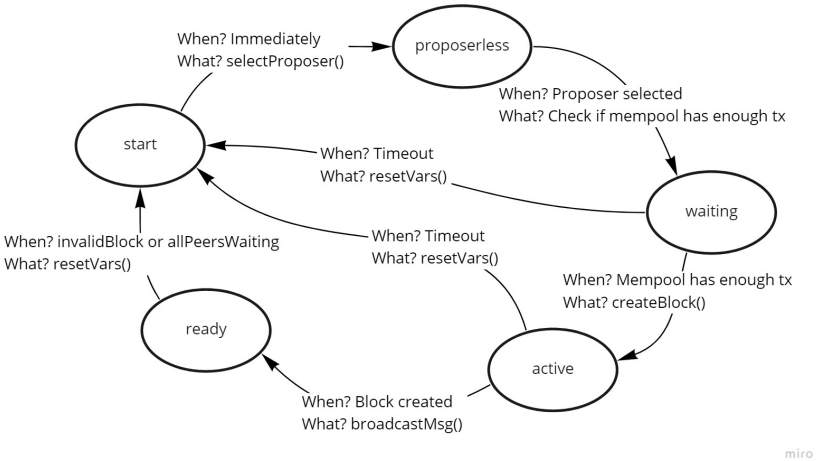

FIGURE 22: State transitions: Tendermint Leader

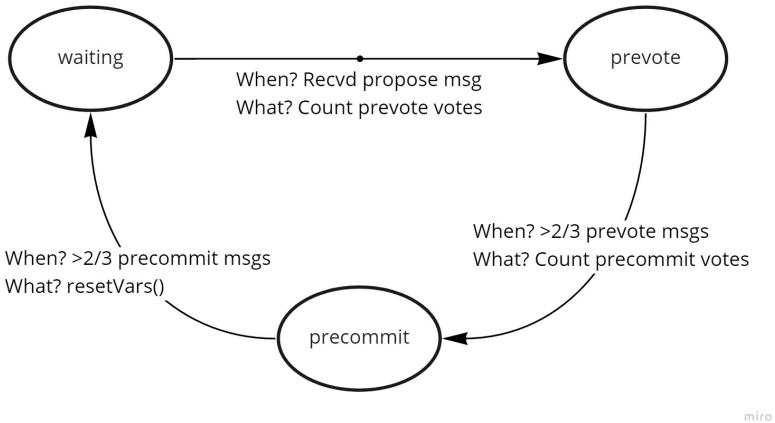

FIGURE 23: State transitions: Tendermint Counter

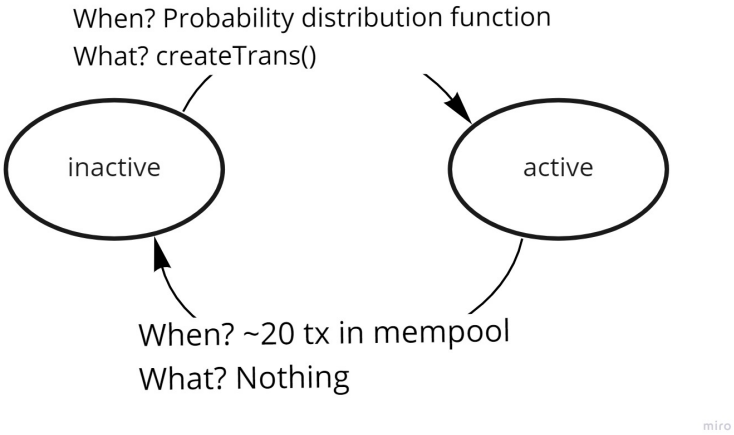

FIGURE 24: State transitions: Clique Client

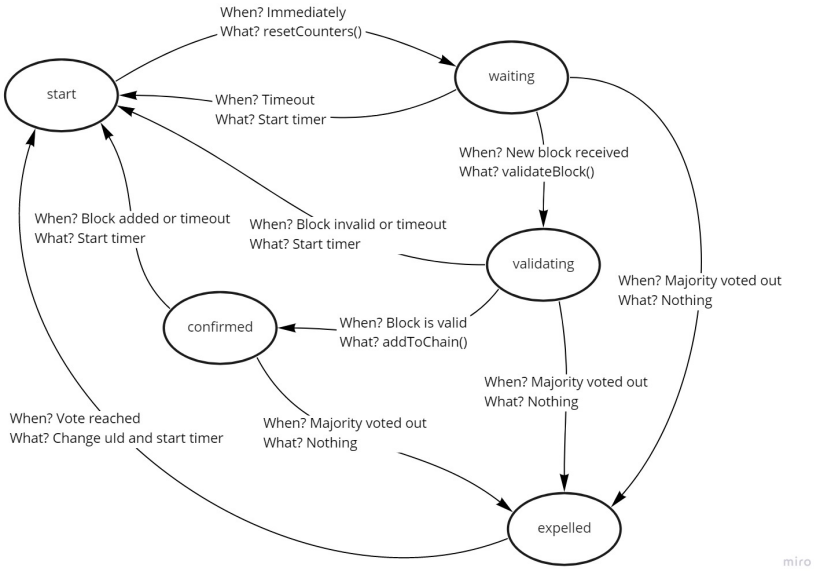

FIGURE 25: State transitions: Clique Peer

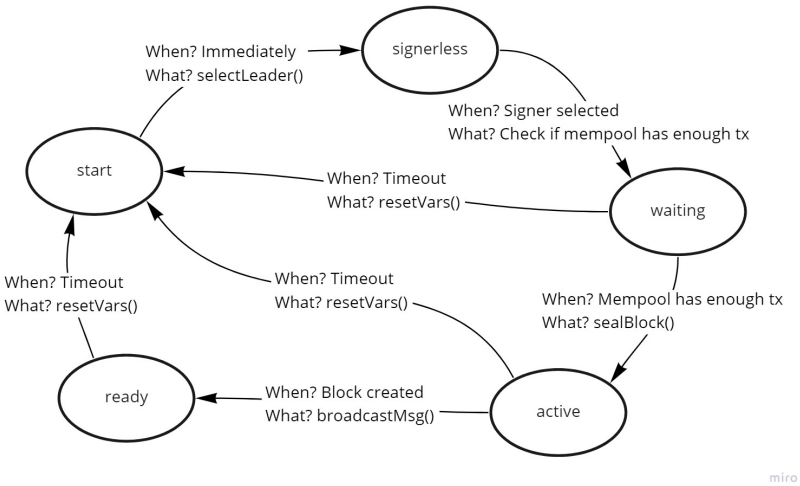

FIGURE 26: State transitions: Clique Leader

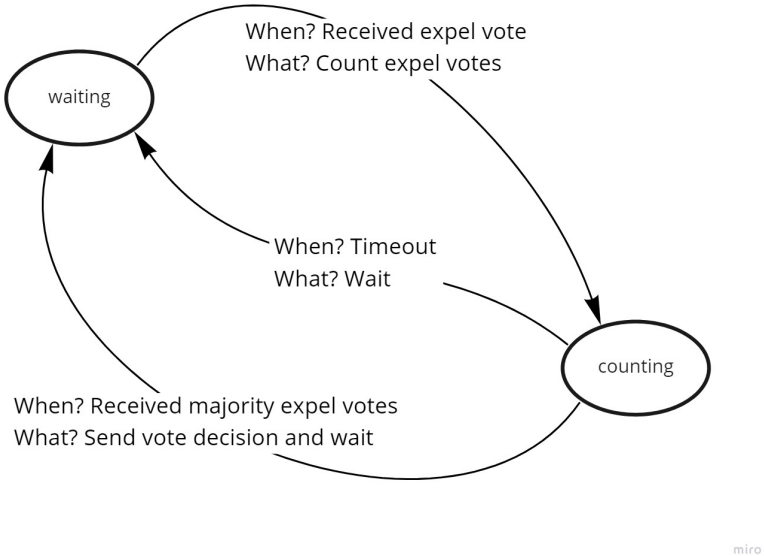

FIGURE 27: State transitions: Clique Counter

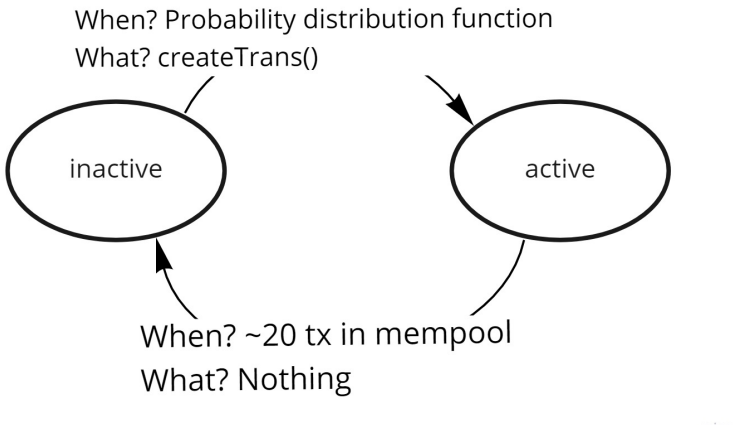

FIGURE 28: State transitions: Raft Client

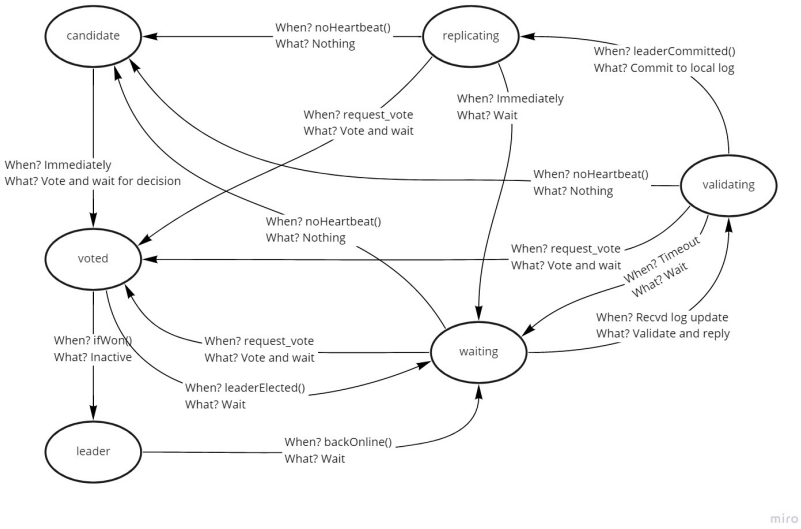

FIGURE 29: State transitions: Raft Peer

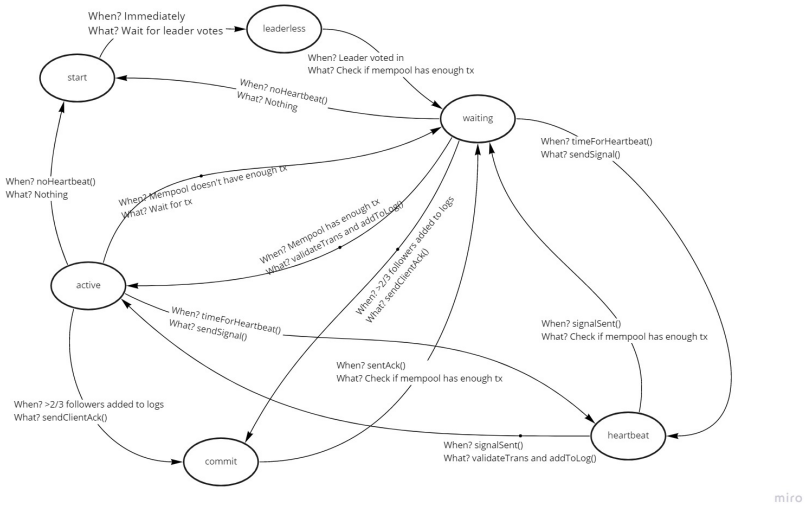

FIGURE 30: State transitions: Raft Leader

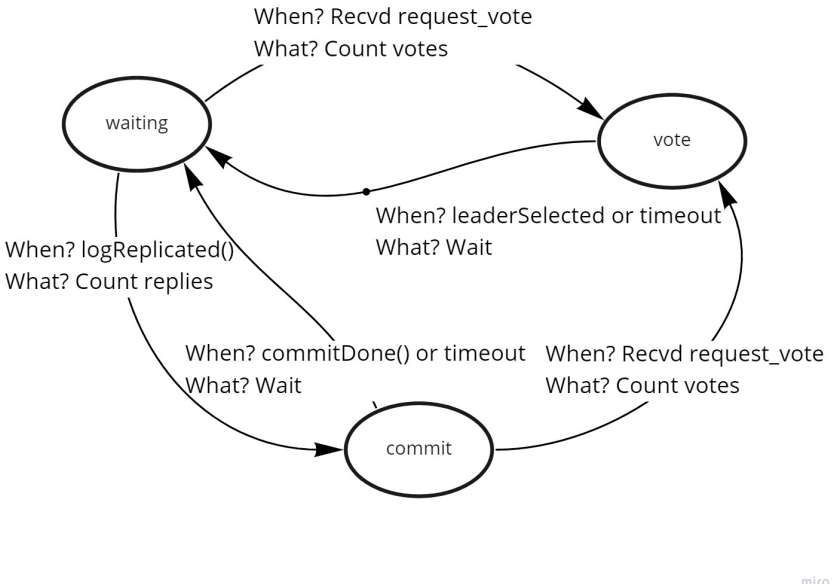

FIGURE 31: State transitions: Raft Counter

REFERENCES

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, Jul. 1982. [Online]. Available: https://doi.org/10.1145/357172.357176

[2] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, p. 288–323, Apr. 1988. [Online]. Available: https://doi.org/10.1145/42282.42283

[3] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. USA: USENIX Association, 1999, p. 173–186.

[4] K. P. How does distributed consensus work? [Online]. Available: https://medium.com/s/story/lets-take-a-crack-at-understanding-distributed-consensus-dad23d0dc95

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[6] C. Walters, "Blockchain consensus encyclopedia," 2018. [Online]. Available: https://github.com/cedricwalter/blockchain-consensus

[7] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," 2019.

[8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference*, 2014.

[9] M. S. Ferdous, M. J. M. Chowdhury, M. A. Hoque, and A. Colman, "Blockchain consensus algorithms: A survey," 2020.

[10] I. Rõžakov, "A modest comparison of blockchain consensus algorithms," 2019. [Online]. Available: http://essay.utwente.nl/78909/

[11] Y. Hao, Y. Li, X. Dong, L. Fang, and P. Chen, "Performance analysis of consensus algorithm in private blockchain," 06 2018, pp. 280–285.

[12] P.-Y. Piriou and J.-F. Dumas, "Simulation of stochastic blockchain models," 09 2018, pp. 150–157.

[13] A. Asgaonkar, P. Palande, and R. S. Joshi, "Is the cost of proof-of-work consensus quasilinear?" ser. CoDS-COMAD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 314–317. [Online]. Available: https://doi.org/10.1145/3152494.3167978

[14] B. Ampel, M. Patton, and H. Chen, "Performance modeling of hyperledger sawtooth blockchain," in *2019 IEEE International Conference on Intelligence and Security Informatics (ISI)*, 2019, pp. 59–61.

[15] A. Ahmad, M. Saad, J. Kim, D. Nyang, and D. Mohaisen, "Performance evaluation of consensus protocols in blockchain-based audit systems," in *2021 International Conference on Information Networking (ICOIN)*, 2021, pp. 654–656.

[16] S. D. Angelis, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, and V. Sassone, "Pbft vs proof-of-authority: applying the cap theorem to permissioned blockchain," in *Italian Conference on Cyber Security (06/02/18)*, January 2018. [Online]. Available: https://eprints.soton.ac.uk/415083/

[17] B. D. Ley-Borrás, Roberto; Fox, "Using probabilistic models to appraise and decide on sovereign disaster risk financing and insurance," 2015. [Online]. Available: https://openknowledge.worldbank.org/handle/10986/22237

[18] Z. Duan, H. Mao, Z. Chen, X. Bai, K. Hu, and J.-P. Talpin, "Formal modeling and verification of blockchain system," in *Proceedings of the 10th International Conference on Computer Modeling and Simulation*, ser. ICCMS 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 231–235. [Online]. Available: https://doi.org/10.1145/3177457.3177485

[19] A. Gopalan, A. Sankararaman, A. Walid, and S. Vishwanath, "Stability and scalability of blockchain systems," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, pp. 1–35, 06 2020.

[20] N. Papadis, S. Borst, A. Walid, M. Grissa, and L. Tassiulas, "Stochastic models and wide-area network measurements for blockchain design and analysis," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 2546–2554.

[21] H. Foundation. Hyperledger caliper. [Online]. Available: https://github.com/hyperledger/caliper

[22] A. Hartmanns and H. Hermanns, "The modest toolset: An integrated environment for quantitative modelling and verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 593–598.

[23] G. Hileman and M. Rauchs, "2017 global blockchain benchmarking study." [Online]. Available: https://ssrn.com/abstract=3040224orhttp://dx.doi.org/10.2139/ssrn.3040224

[24] G. Denaro, A. Polini, and W. Emmerich, "Early performance testing of distributed software applications," vol. 29, 01 2004, pp. 94–103.

[25] R. Ghazali, J. O. Haryanto, W. h. Utomo, A. S. Santoso, R. Nugraha, and B. Asgha, "Exploring the drivers of mobile based peer to peer lending application service quality in indonesia," in *2019 International Conference on Sustainable Engineering and Creative Computing (ICSECC)*, 2019, pp. 343–348.

[26] S. A. Demurjian, D. K. Hsiao, D. S. Kerr, R. C. Tekampe, and R. J. Watson, "Performance measurement methodologies for database systems," in *Proceedings of the 1985 ACM Annual Conference on The Range of Computing: Mid-80's Perspective: Mid-80's Perspective*, ser. ACM '85. New York, NY, USA: Association for Computing Machinery, 1985, p. 16–28. [Online]. Available: https://doi.org/10.1145/320435.320446

[27] D. Wybranietz and D. Haban, "Monitoring and performance measuring distributed systems during operation," in *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 197–206. [Online]. Available: https://doi.org/10.1145/55595.55618

[28] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1085–1100. [Online]. Available: https://doi.org/10.1145/3035918.3064033

[29] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014, accessed: 2016-08-22. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[30] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi, "Smart Contract Security: A Software Lifecycle Perspective."

[31] H. Foundation. Hyperledger sawtooth. [Online]. Available: https://github.com/hyperledger/sawtooth-core

[32] Cosmos. Cosmos sdk. [Online]. Available: https://github.com/cosmos/cosmos-sdk

[33] Ethereum. Go ethereum. [Online]. Available: https://github.com/ethereum/go-ethereum

[34] M. C. Y.T. Lin, A. Chen. Istanbul byzantine fault tolerant consensus protocol. [Online]. Available: https://github.com/ethereum/EIPs/issues/650

[35] H. Foundation. Hyperledger fabric. [Online]. Available: https://github.com/hyperledger/sawtooth-core

[36] K. S. Narendra and M. A. L. Thathachar, "Learning automata - a survey," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-4, no. 4, pp. 323–334, 1974.

[37] L. Desgeorges, P.-Y. Piriou, T. Lemattre, and H. Chraibi, "Formalism and semantics of pycatshoo: A simulator of distributed stochastic hybrid automata," *Reliability Engineering System Safety*, vol. 208, p. 107384, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0951832020308711

[38] Docker. Docker containerization software. [Online]. Available: https://www.docker.com/

[39] Locust. Locust. [Online]. Available: https://github.com/locustio/locust

[40] A. Ledenev. Pumba. [Online]. Available: https://github.com/alexei-led/pumba

[41] H. Fabric. High-throughput network. [Online]. Available: https://github.com/hyperledger/fabric-samples/tree/release/high-throughput

VITA AUCTORIS

NAME:	Shiv Sondhi
PLACE OF BIRTH:	Mumbai, Maharashtra, India
YEAR OF BIRTH:	1996
EDUCATION:	Manipal Institute of Technology, Manipal, Karnataka, Bachelor of Technology in Computers And Communication Engineering, 2014-2018
	University of Windsor, Windsor, Ontario, Master of Science in Computer Science, 2019-2021