

Decentralized Consensus Algorithm For Scalable Blockchains

By

T. M. Amith

2015CS011

This dissertation is submitted to the University of Colombo School of Computing

In partial fulfillment of the requirements for the

Degree of Bachelor of Science Honours in Computer Science

University of Colombo School of Computing

35, Reid Avenue, Colombo 07,

Sri Lanka

July 2020

Declaration

I, T. M. Amith (15000117) hereby certify that this dissertation entitled decentralized consensus algorithm for scalable blockchains is entirely my own work and it has never been submitted nor is currently been submitted for any other degree.

Candidate Name: T. M. Amith

.....
Date

.....
Student Signature

I, Dr. Kasun De Zoysa, certify that I supervised this dissertation entitled decentralized consensus algorithm for scalable blockchains conducted by T. M. Amith in partial fulfillment of the requirements for the degree of Bachelor of Science Honours in Computer Science.

.....
Date

.....
Signature of Supervisor

Abstract

Blockchain is a decentralized distributed ledger. Today blockchain technology is one of the most researched and adapted to modern systems due to its security advantage. The digital currency Bitcoin is the living proof of how much blockchain technology can provide reliability and trust into any system since it is not managed or governed by any organization or person. But when looking into such systems which are decentralized, there are shortcomings like scalability.

This research proposes a consensus algorithm for private blockchains that can scale with increasing number of ordering nodes. Currently almost all blockchains have a threshold limit of how much transactions that can be processed in a given amount of time. Blockchains that are public and include massive amount of users in the network process transactions at a very slow rate. Bitcoin and Ethereum process 7 and 15 transactions per second respectively. VISA processes around 25000 transactions per second. Therefore, blockchains are still not at a level that can replace the traditional financial systems. This case is similar to that of private blockchains. Even though they are faster than public blockchains, then again the requirement for in systems that limited in the same manner are even greater than what current blockchain implementations can provide.

The Canopus consensus algorithm proposed here promises scalability to increasing amount of ordering nodes in the network. It is tested on the Hyperledger Fabric framework version 1.4. A sample smart contract is instantiated for all the tests. The performance was evaluated for different amounts of ordering nodes.

Preface

Through this research, a new consensus algorithm is proposed that can be adapted instead of Raft and Kafka currently available in Hyperledger Fabric. It is scalable as well as crash fault tolerance which is the requirement for many business organizations for their systems. The implementation of the consensus algorithm is done over the Hyperledger Fabric source code. The code and design of this algorithm and the performance evaluation is solely based on my work. The design of the algorithm is taken from the Canopus research paper with no implementation details provided or the code that the authors have used to carry out their research. The adaptation of Canopus consensus algorithm for private blockchains have not been proposed by any other related work. Evaluation model for the performance evaluation was devised by referring related performance evaluation studies by myself under supervision.

Acknowledgement

I would like to express my sincere gratitude to my research supervisor, Dr. M.D.J.S. Goonetillake, Senior Lecturer at the University of Colombo School of Computing for the continuous guidance and supervision provided to me throughout the research. I would also like to extend my sincere gratitude to Dr. Kasun De Zoysa, Senior Lecturer at the University of Colombo School for providing feedback on my research proposal and interim evaluation and continuous guidance and motivation to do a better research. I would also take the opportunity to acknowledge the assistance provided by Dr. H. E. M. H. B. Ekanayake as the final year computer science project coordinator.

I would like to dedicate this work to my parents who have helped me immensely throughout this journey. I would also like to acknowledge the lecturers and batch mates who helped me in numerous ways and contributing to my knowledge and experience and for a successful research.

Table of Contents

Contents

Declaration	ii
Abstract	iii
Preface	iv
Acknowledgement	v
Table of Contents	vi
List of Figures.....	ix
List of Tables	x
List of Acronyms	xi
Chapter 1 - Introduction	1
1.1 Background to the research	1
1.2 Justification for the Research	2
1.3 Research Problem & Research Questions	3
1.4 Methodology.....	3
1.5 Outline of the Dissertation.....	5
1.6 Delimitations of Scope	5
Chapter 2 - Literature Review	6
2.1 Introduction	6
2.2 Blockchain.....	6
2.3 Types of blockchains	7
2.4 Blockchain Scalability	8
2.5 Scalability Metrics.....	10
2.6 Scalability Bottlenecks	11
2.7 Network Plane	12
2.8 Consensus Plane	12
2.9 Storage Plane	12
2.10 View Plane.....	13
2.11 Side Plane	13
2.12 Blockchain Trilemma	13
2.13 Consensus	14
2.14 Crash Tolerant Consensus	15
2.14.1 Paxos.....	15
2.14.2 Kafka / ZooKeeper	16

2.14.3	Raft	17
2.14.4	Quorum Chain	17
2.15	Byzantine Consensus	18
2.15.1	Proof of Work	18
2.15.2	Proof of Stake	19
2.15.3	Delegated Proof of Stake (DPoS)	19
2.15.4	Practical Byzantine Fault Tolerance(PBFT)	19
2.15.5	BFT-SMaRT	20
2.16	Overview of Blockchain Scalability	21
2.16.1	Permissionless Blockchains	21
2.16.2	Permissioned Blockchains	22
2.17	Consensus Protocols Not Yet Adapted to Blockchains	23
2.18	Canopus	24
2.19	Integrate Canopus to a Blockchain	25
2.20	Hyperledger Fabric	25
2.21	Hyperledger Fabric Architecture	26
2.21.1	Execution Phase	27
2.21.2	Ordering Phase	28
2.21.3	Validation Phase	29
Chapter 3	- Design	30
3.1	Introduction	30
3.2	What Canopus Is & Not	30
3.3	Centralization & Decentralization	30
3.4	Exploiting Network Hardware	32
3.5	Leaf Only Trees	32
3.6	Consensus Cycle	33
3.7	Conclusion	29
Chapter 4	- Implementation	30
4.1	Introduction	30
4.2	Define Organizations	30
4.3	Generate Cryptographic Certificates	31
4.4	Define Orderer Related Parameters & Profile	31
4.5	Generate Channel Artifacts	32
4.5.1	Genesis Block	32
4.5.2	Channel Configuration Transaction	33
4.6	Implementing Canopus Consensus	34

4.6.1	Introduction	34
4.6.2	Receiving Transaction Messages.....	34
4.6.3	Queue Messages and Trigger the Start of Consensus.....	35
4.6.4	Distribute Messages.....	37
4.6.5	Calculate The Final State.....	37
4.7	Build Orderer Package & Docker Image.....	38
4.8	Start and Initialize Fabric Network	38
4.9	Summary.....	39
Chapter 5	- Results and Evaluation	40
5.1	Introduction	40
5.2	Evaluation Process.....	40
5.3	Evaluation of Consensus Protocols	41
5.3.1	Evaluation of open-account smart contract	41
5.3.2	Evaluation of transfer-money smart contract	43
5.4	Summary.....	44
Chapter 6	- Conclusion.....	46
6.1	Introduction	46
6.2	Conclusions About Research Questions.....	47
6.3	Limitations and Implications for Further Research	48
References	49
Appendix C	54
Appendix C.1	54
Appendix C.2	55
Appendix C.3	57
Appendix C.4	59
Appendix C.5	60

List of Figures

Figure 3.1 - Illustration of Raft consensus31

Figure 3.2 - LOT in Canopus26

Figure 3.3 - Six nodes arranged in two super-leaves in a LOT of height 2.....27

Figure 3.4 - High level architecture of proposed blockchain network.....28

Figure 4.1 - Command line to build genesis block.....33

Figure 4.2 - Dispatcher.go functions to receive messages from other orderer nodes.....35

Figure 4.3 - Chain struct in chain.go file36

Figure 4.4 - SendSubmit function to distribute messages with other nodes in the orderer service37

Figure 5.1 Transaction throughput of consensus algorithms on open-account smart contract42

Figure 5.2 Latency of consensus algorithms on open-account smart contract42

Figure 5.3 Transaction throughput of consensus algorithms on transfer-money smart contract43

Figure 5.4 Latency of consensus algorithms on transfer-money smart contract.....44

List of Tables

Table 2.1- Differences and similarities of public and private blockchains 8

Table 2.2 - Comparison of features of several permissioned DLT platforms.....23

Table 2.3 Comparison of pluggable ordering services for Hyperledger Fabric28

List of Acronyms

PBFT – Practical byzantine fault tolerance

SBFT – Simplified byzantine fault tolerance

DLT – Distributed ledger technology

PoW – Proof-of-work

DPoS – Delegated proof-of-stake

Chapter 1 - Introduction

1.1 Background to the research

Blockchains have made a huge impact in distributed systems and financial systems after the emergence of cryptocurrency such as Bitcoin. Institutions in healthcare [38], asset tracking [39] utilize private blockchains for the convenience of security and the trust the users and customers have with time developed around this technology. But as in any new technologies that emerge, there are plenty of room for improvement.

One main feature of blockchain with respect to all other traditional distributed systems is that it is decentralized. Decentralization is the process by which the activities of an organization, particularly those regarding planning and decision making, are distributed or delegated away from a central, authoritative body. In blockchain, this is delegated to all the people or users of the blockchain. This is the case in a peer-to-peer network where files are shared among the peers without the need of a central authority. But blockchain is a distributed ledger. Unlike a peer-to-peer network, there is the need of confirming the legitimacy of each peer's ledger and keep the ledger consistent throughout the network. This is where consensus comes in to play. The consensus algorithm plays a major role in blockchains and is the main focus of this research.

Depending on whether a blockchain is public where any amounts users can join the network or whether it is private where only certain amount of users are allowed to join the network determined by the organizations that deploy it, there are 2 classes of consensus algorithms. Proof of work, proof of stake, delegated proof of stake, proof of elapsed time are some consensus algorithms used in public blockchains. They are very slow in terms of transactions that can be processed given a time frame. It does not spam more than 1000 transactions per second.

On the other hand, consensus algorithms such as PBFT, SBFT, Raft, Kafka, solo that are used in private blockchains have more processing power. But in most scenarios, the scalability of these algorithms is questionable at best. The upper threshold can be the reason a business organization might abandon the use of blockchains because there is no

capability to scale. Thus it is worthwhile to explore algorithms that can overcome this barrier and make blockchain suitable for huge applications that scale massively.

1.2 Justification for the Research

A Distributed Ledger Technology (DLT) network is a collection of interconnected nodes where, each node maintains a copy of the same database, called the ledger. In DLT, there is no centralized database which is controlled or administered by a central party that is trusted by every participant. The process of updating the distributed ledger requires exchanging transaction information between nodes, achieving distributed consensus among nodes, followed by adding the validated transaction as a new ledger entry.

If blockchain is the underlying database structure of the ledger, the ledger could be identified as a hash chain over blocks. Thus, during the last step of updating the distributed ledger ('adding the validated transactions as a new ledger entry'), validated transactions are grouped into blocks and appended to the ledger (i.e. the blockchain).

Distributed Ledgers have several advantages over traditional (centralized) databases. DLT provides a full audit trail of information history, provides accessibility to a common view of information to all nodes at the same time and it is impossible to make unauthorized changes to the distributed ledger.

There are two main types of distributed ledgers, namely, permissionless distributed ledgers and permissioned distributed ledgers. A permissionless DLT network is accessible to anyone, i.e. all participants are public nodes, while a permissioned DLT network contains an authorized consortium of participants. Procedure of obtaining distributed consensus in an permissionless DLT network is through "Proof of Work" (PoW) mining, while, in a permissioned DLT network distributed consensus is obtained through validation by a selected subset of 'trusted validating nodes'.

Blockchain has gained much popularity due to its key features and advantages over traditional DLT technologies. However due to limitations of blockchains in terms of scalability, the adaptation of blockchain to many production applications is not considered. Here we try to address this issue by providing a solution for scalable blockchains which in turn increase the application domain of blockchains.

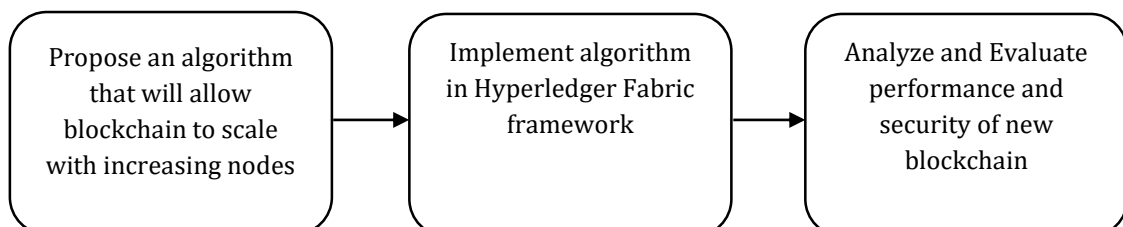
1.3 Research Problem & Research Questions

Is a truly scalable consensus algorithm feasible within blockchain?

- What are the bottlenecks of achieving scalable blockchains?
- How does consensus in blockchains limit blockchain performance?
- Is there a consensus protocol that could overcome the bottlenecks in current consensus in blockchains?
- What is the performance of proposed consensus algorithm in blockchains?

Answers to research questions will be obtained with the completion of listed research objectives. All questions will be focused on the scalability factor of blockchain. All decentralized distributed systems like blockchain faces the drawbacks of scalability trilemma. Scalability trilemma describes that a system can never achieve the 3 most important aspects of a distributed system. They are fast or scalability, decentralization and security. Currently either one of these properties get compromised regardless of the approach and consensus that is applied. Blockchain which also prominently faces this issue may be able to overcome the barrier if the research becomes a success. Depending on the findings on the literature review, the consensus algorithm will be selected or modified in order to provide the required features to the blockchain. Here by adopting a different ordering protocol, it is expected to provide the scalability and security of a close to ideal system.

1.4 Methodology



Distributed consensus algorithms were researched from the early 1990s. State machine replication placed the foundation for the first ever applicable distributed consensus algorithm supporting atomic broadcast. Consensus traditionally means the task of

reaching an agreement on one single request whereas atomic broadcast [40] provides agreement on a sequence of requests needed for state-machine replication. Both terms will imply the same thing here but the most appropriate technical term is atomic broadcast. When the requirement arose to build permissioned blockchains for private business organizations, the most referred consensus protocols to build blockchains were Paxos and Viewstamped Replication [25]. On this family of protocols, permissioned blockchains that can tolerate both crash fault and fault tolerance consensus was built. Practical Byzantine Fault Tolerance, Zab protocol from Zookeeper used in Apache Kafka, Raft algorithm used in Hyperledger Fabric all apply the above mentioned protocols. However, these were meant to build resiliency and guarantee safety and liveness to the system. Inherent nature that followed when guaranteeing these factors was that it did not allow any system that applies consensus algorithms to be scalable. Before selecting a suitable algorithm that could be scalable, it requires to do a broad study in the area of consensus in distributed ledger technology. The main factors that will be considered when selecting a suitable consensus algorithm are that it should be at least crash fault tolerance and byzantine fault tolerance to some extent and it should provide a solution to the scalability of distributed systems with consensus.

The selected algorithm then will be implemented in Hyperledger Fabric. The main reason for selecting Fabric is the capability of pluggable consensus. The total ordering of the requests is done separately from the execution and validation phases. This modular architecture has attracted many researchers to test out various consensus mechanisms using Hyperledger Fabric like in [30]. This research will be similar to such an approach and the Fabric platform provides the best features to work on it. Also this is made possible because Fabric is open source. A thorough walkthrough of the code especially of the consensus module should be carried out. Then adopting the techniques used in building nodes and connecting with peers and clients, develop the Canopus protocol and have this consensus set up instead.

Finally, the implemented blockchain is evaluated on the amount of transactions which can reach consensus for a given period of time by having varied amount of orderer nodes. Organizations that grow will always require more performance in their systems to accommodate the increasing demand. Therefore, when such a situation rise and they decide to scale their blockchain system, it should never slow down the system. Otherwise

they would tend to look for alternate solutions. Providing a suitable blockchain framework to those businesses is the contribution through this research.

1.5 Outline of the Dissertation

The dissertation will be structured as follows. Chapter two will present a detailed review regarding the history and the evolution of consensus algorithms used in distributed systems as well as details regarding the blockchain framework that will be used for this research to implement the proposed consensus algorithm. Chapter three will represent the design of the proposed algorithm and how it will accommodate within the selected blockchain framework. Chapter four explains the implementation details of the blockchain framework and the consensus algorithm. Chapter five will include the evaluation of the of the implemented blockchain. The final chapter provides a conclusion for the thesis with prospects for future work.

1.6 Delimitations of Scope

In this research project, only the consensus of the blockchain is considered. No any other optimizations will be attempted. With the performance evaluation, crash fault tolerance will also be tested in the research project as it is necessary to evaluate whether the blockchain is compromised by the proposed consensus algorithm.

Chapter 2 - Literature Review

2.1 Introduction

This chapter will provide a detailed review of blockchains, types of blockchains, scalability factors of blockchains, bottlenecks, consensus algorithms used in blockchains, and consensus not adapted to blockchains. In the review, the first 3 research objectives are addressed and answered.

2.2 Blockchain

A distributed ledger is a database that is shared and synchronized across multiple sites, institutions or geographies. The main advantage in a security point of view of having such a database spread across multiple nodes makes it difficult for a cyber-attack to damage all the replicated records [1]. A blockchain is such a distributed ledger which the nodes that belong to the network maintain and update the records without any central authority [2]. Blockchains not only maintain such a ledger but they are capable of executing smart contracts over their network. Smart contracts are computer programs that can be consistently executed by a network of mutually distrusting nodes, without the intervention of any authority [3]. This nature of smart contracts creates their resilience to tampering and makes it favorable for many use cases such as executing financial transactions and transferring confidential documents. Therefore, blockchain is a service that maintains a state which is the distributed ledger and clients can invoke functions within the network that can update the overall state and provide a response [4] through smart contracts. All the nodes in the service have some stake for when the state is updated. On the assumption that no node is trustworthy in the network, there are certain protocols the blockchain implements in order to achieve a unanimous decision on the final updated state of the network.

2.3 Types of blockchains

Blockchain platforms can be mainly classified into 2 main types.

1. Permissionless blockchain
2. Permissioned blockchain

Permissionless blockchains as the name implies are open systems which are publicly available [5]. Any node which represents a certain user of the blockchain can invoke transactions to view and update the state as well as participate in the process of advancing and validating the transactions the network receives. Here the number of nodes can vary and extend to very large numbers. Since this is publicly accessible, most nodes are anonymous and must be assumed as untrustworthy [5]. The first ever permissionless blockchain, Bitcoin is a cryptocurrency that was first released in 2009[6]. The whitepaper [6] was published in October 2008 by Satoshi Nakamoto who still remains unknown. Bitcoin is the first digital cash that was based on a truly decentralized system [7]. But Bitcoin did not allow the use of its technology beyond cryptocurrency. Ethereum is the first project that built a generalized technology where an end-developer can create their own smart contract and deploy over the blockchain network [8]. Besides Bitcoin and Ethereum, there are many more alternative platforms that have been developed throughout the years which implement the cryptocurrencies or smart contracts [9, 10, 11, 12]. The number of cryptocurrencies hosted on <https://coinmarketcap.com/> have increased from 0 to over 5000 since 2012 which show the massive popularity and investment that is going into blockchain technology.

A permissioned blockchain in contrast to permissionless blockchain is operated by a known set of organizations or entities which is sometimes called a consortium blockchain where the members of it who become the stakeholders of the business operate the network [2]. There are separate systems within these types of blockchains that identify the nodes that can control and update the state of the network. A private blockchain can be considered a network maintained by a single body of authority [2]. Generally, the number of nodes in permissioned blockchains are small compared to permissionless blockchains. Similar in case with permissionless blockchains, there are a large number of permissioned blockchains that are available and provide similar features with few differences [13, 14, 15, 16, 17].

Table 2.1- Differences and similarities of public and private blockchains

Public Blockchain	Permissioned Blockchain
Function as an append-only ledger	Function as an append-only ledger
Each network node in both these blockchains has a complete replica of the ledger	Each network node in both these blockchains has a complete replica of the ledger
Validity of a record is verified	Validity of a record is verified (but the mechanism maybe different)
Rely on numerous users to authenticate edits	Rely on numerous users to authenticate edits
Anyone can take part by verifying and adding data to the blockchain	Only authorized entities can participate and control the network
Completely decentralized	Somewhat centralized around the entities that own the blockchain
Throughput is lesser	Throughput is higher
Not scalable	Scalable to a certain limit
Consume more energy (in terms of computation power)	Consume less energy
No one knows who each validator is and this increases the risk of potential collusion	No chance of minor collusion

2.4 Blockchain Scalability

According to Wikipedia, scalability is a property of the system to handle increasing amount of workload by certain techniques. On a very abstract level, there are mainly 2 ways to scale a system. The first is vertical scaling where the system performance is increased by installing more powerful hardware (more RAM, storage, etc.). The other is horizontal scaling by adding more machines into the system and dividing the workload among the connected machines. This is the basic notion of distributed systems. Therefore, it concerns blockchains as well.

Bitcoin, which was the first ever blockchain platform that became publicly available, showed the world its ability to carry out digital transactions secure and in a transparent manner without any authority governing over the network [18]. Currently according to coindesk.com, the value of a single Bitcoin is approximately 9400 US dollars and value of a single Ether is over 180 US dollars. These values are so high due to the high demand of these currencies. The increasing adoption of cryptocurrencies has raised many concerns regarding scalability and the cryptocurrency community and researchers have been working on many techniques to improve scalability [19]. Looking at some statistics, Bitcoin takes 10 minutes or longer to confirm transactions, that is a node becomes able to create a block and transfer this block throughout the network to more than 50% of the users which achieves a maximum throughput of 7 transactions per second. In contrast, looking at a mainstream payment service like Visa or Mastercard credit, it confirms transactions within seconds and reach maximum throughput of 56,000 transactions per second [19]. If we were to move to a complete decentralized system where no governing body as Visa will maintain the payment service, the blockchain will require to match these throughput numbers for it to be considered a viable alternative. Therefore, the major drawback can be seen in blockchains which is the lack of scalability.

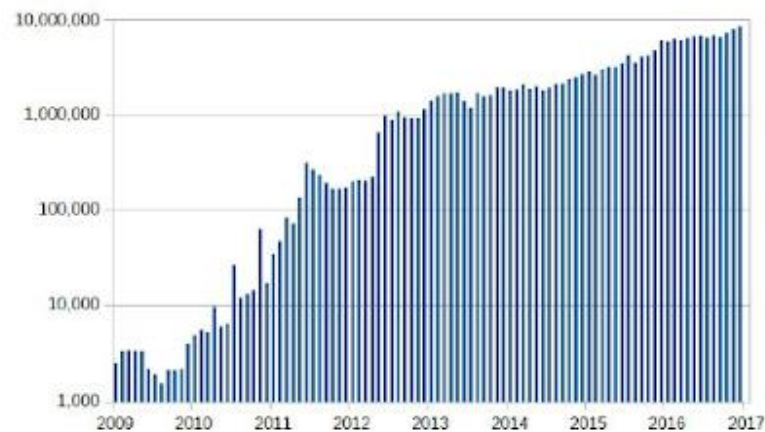


Figure 1 : A graph denoting the increase in bitcoin transactions over the last 10 years.
Source : blockgeeks.com

There are mainly 2 types of scaling when concerned with blockchains.

1. Offchain scaling
2. Onchain scaling

Offchain scaling is where a solution is developed in order for small and frequent transactions take place over low-tier blockchain instances, executed parallel with the main blockchain. An example of offchain scaling is bitcoin lightning protocol [18]. The protocol sets up a network which effectively creates a layer on top of bitcoin, enabling fast and cheap transactions which can settle to the bitcoin blockchain. This network is responsible for creating channels between users or mainly between wallets and the 2 parties can transfer any amount of Bitcoin without touching the information on the blockchain. This method is comparatively less secure and therefore used largely for small or micro transactions which are lower risks to the concerned parties. Therefore, the main concept in offchain scaling is taking the workload out of the main blockchain and executing in a different network of nodes without compromising security.

Directly modifying the core blockchain design to increase the performance and throughput is called onchain scaling [20]. Unlike in offchain, where the main blockchain design remains unchanged, onchain scaling tends to change a certain core mechanism to increase performance that might change the whole architecture of the blockchain. Some techniques derive a whole new blockchain which is deployed separately. A good example here is Bitcoin-NG [21]. It takes the same trust model as Bitcoin but decouples leader election which is achieved through proof-of-work [6] from transaction serialization. A leader in the Bitcoin-NG network appended multiple transactions to the blockchain for the duration of the epoch which ends when a new leader is elected in contrast to only capable of appending a single block like in Bitcoin. Onchain scaling can provide massive benefits when compared to offchain scaling but it is the most difficult task out of the 2 types.

2.5 Scalability Metrics

There are mainly 2 metrics that directly relate to blockchain scalability.

1. Transaction throughput
2. Latency

Transaction throughput is the maximum rate at which the blockchain can execute and validate transactions i.e. transactions that can be processed per second [20]. Latency is

the time it takes to confirm that a transaction has been verified and appended to the blockchain [20]. Taking a look at Bitcoin, transaction throughput is a relation between the block size and the inter-block interval. The block size is the maximum allowed size of a block. The inter-block interval is the amount of time given to receive and append transactions to the block. With a block size of 1 Megabyte and a block interval of 10 minutes, the transaction throughput is around 7 transactions per second [20]. Ethereum due to the gas limit imposed on each block which is around 10,000,000 gas, it gains a maximum throughput of 20 transactions per second. Considering a permissioned blockchain like Hyperledger Fabric, the transaction throughput with a block size of 2 Megabytes and block interval of 500ms is an average between 3000-3500 transactions per second [13]. The latencies of each of these cases is the same as the inter-block interval but might differ in some other blockchains.

2.6 Scalability Bottlenecks

According to [19] the blockchain architecture can be decomposed to 5 abstract layers called planes that directly or indirectly affect the scalability of it. How each layer will contribute to scalability and what are the bottlenecks in the current systems in each layer will be discussed. Other than these 5 planes, there is another technique to increase the performance of the blockchain called parameter tuning [19]. Here we adjust certain values that can be easily updated in the blockchain in order to improve performance. The maximum block size and the inter-block interval are the 2 main parameters that can be fine-tuned. Since there is no design change or change in execution of processing transactions, it only provides very limited performance boost and it is not sufficient for the scaling requirement we are trying to accomplish [19].

The 5 main planes of a blockchain system are:

1. Network Plane
2. Consensus Plane
3. Storage Plane
4. View Plane
5. Side Plane

2.7 Network Plane

The network plane describes how the transaction messages get propagated among the nodes in the blockchain. In permissionless blockchains like Bitcoin, the messages are broadcasted to every other node in the network [6]. This is not the case in permissioned blockchains. Considering Hyperledger Fabric, the transaction messages are sent to peer nodes, then upon receiving their endorsements it is sent to orderer nodes. After consensus by the ordering nodes, the result block is sent back to peers to validate and update the ledger [13]. Decreasing the network latency and traffic can lead to better performance of transaction execution in the blockchain.

2.8 Consensus Plane

The main function of the consensus plane is to globally decide on the order of transactions to be processed in the blockchain. This plane ingests messages arriving from the network plane and sends the ordered messages to update the system ledger [19]. For example, in Bitcoin, the consensus plane is responsible for mining blocks and integrating it to the blockchain ledger. In this research we will be focusing on the consensus plane, the various types of consensus protocols available, their limitations and how to improve the consensus mechanism thus improving scalability.

2.9 Storage Plane

The storage plane is the blockchain database. It contains the blockchain ledger and other states that are required to be stored. This plane is also responsible for writing operations that modify the ledger and respond to read requests from other entities in the system like peer nodes [19]. Bitcoins' storage includes the whole entire ledger which takes around 4 days to process when a read operation is executed. Having to deal with the entire ledger is inefficient and unnecessary in most cases. Solutions like sharding where only certain parts of the ledger are stored by a single node are being researched [19]. In the case of

Hyperledger Fabric, only the peer nodes save the ledger of the blockchain and the ledger can be queried through smart contracts [13].

2.10 View Plane

A state of a blockchain is the execution of the whole transaction history in the ledger. If a read operation or write operation needs to be executed, this state should always be calculated. This is a completely unnecessary operation and it is overcooked using the view plane. A view can be considered as a data structure derived from the full ledger where the state is calculated by processing all the transactions in the ledger [19]. This view allows any entity to access the state without reading any part of the ledger which is a key performance requirement in blockchains today. When Bitcoin started out, it did not implement any sort of view. Later a view called unspent transaction outputs(UTXO) was implemented that represented the state of the Bitcoin blockchain. But for a newcomer to the network, the node will require downloading the whole ledger and build the UTXO which takes around 4 days [19]. Ethereum, Fabric and other blockchains that support smart contracts can define and request the state of the ledger.

2.11 Side Plane

The side plane represents the off-chain functionalities that can be implemented over a blockchain which provide off-chain scalability. As discussed in chapter 2.3, some workloads can be executed over a separate network to increase the performance of the underlying blockchain. The best example here is the lightning network for Bitcoin [18].

2.12 Blockchain Trilemma

Thus, the ‘Scalability Trilemma’ is defined by the founder of Ethereum, Vitalik Buterin. According to Buterin, the trilemma is developing a blockchain technology that offers

security, decentralization, and scalability without compromising any one of them. It is known that currently, blockchain technology can only offer two of the three variables [22]. Bitcoin is the example that achieves decentralization and security.

2.13 Consensus

Consensus in literature means arriving at a general agreement for a single request. The technical term for arriving at an agreement among a set of nodes on a sequence of requests needed for state-machine replication is called atomic broadcast [2]. Due to the close resemblance among the two terms, the term consensus is used in the context of blockchains to mean atomic broadcast. State-machine replication is the concept or algorithm to maintain the state of machines when they are replicated for reasons such as scalability and distributed computing. According to [23] there are mainly 2 tasks for reaching and maintaining consensus among the distributed nodes.

1. A deterministic state machine that implements the logic to be replicated.
2. A consensus protocol to disseminate requests among the nodes such that each node executes the same sequence of requests on its instance of the service.

Consensus in blockchains was first introduced by Bitcoin. In Bitcoin's proof of work [6] consensus protocol, the consensus on one shared ledger is based on voting among the nodes through their CPU power. The nodes work on extending the longest chain and reject invalid blocks which are the shorter branches of the chain. This is called the long chain rule [6]. With the increase of popularity of Bitcoin, existing consensus and replication mechanisms received renewed attention and research were done to see how they can contribute for more performance and scalability in blockchain.

There are 2 types of consensus algorithms according to the type of tolerance they provide in a network where crashing of nodes and malicious nodes may or may not be present.

2.14 Crash Tolerant Consensus

Crash tolerant consensus as the term means is consensus algorithms that are tolerant to crashes among the nodes in the network. There is a limit to the amount of nodes that are allowed to be crashed or offline during the consensus process. One of the most prominent ways to implement consensus in distributed systems containing 'n' nodes and have a probability of 'n/2' nodes crashing as the worst case, is the family of protocols known as Paxos [24] and Viewstamped Replication(VSR) [25]. The core mechanisms in these protocols are very similar and are implemented in many cloud services and distributed systems today [2]. Some prominent implementations of protocols under this family are Zab protocol inside ZooKeeper [26, 27] which is used in a popular stream processor service Apache Kafka and Raft [28] protocol which is the latest addition of consensus protocols which was developed in the aim of simplifying the process represented in Paxos.

The consensus process in this family of protocols progress through a sequence of views or epochs where each epoch requires a leader node who will overlook the process. If the leader crashes, a new leader is appointed through a voting system and moves to the next epoch. During a single epoch, the leader will broadcast the messages among the nodes in the network in a single agreed sequence maintaining total order among all the nodes in the network [2].

2.14.1 Paxos

Paxos [24] is the most prominent and well known due to its practicality and easy implementation. Paxos is used in many distributed systems like cloud computing, databases and even as a base framework in crash tolerant consensus in blockchains. Looking into consensus algorithm at a fundamental level, it ensures that a single one among the proposed values is chosen. Paxos guarantees that one value is chosen and operates correctly in a non-byzantine fault environment. Paxos protocol mainly consists of 3 agents. They are proposers, acceptors, and learners. When applied to a specific system, the names of the agents change but their function remain the same. [24]

approaches the solution step by step clearly explaining the reason behind the chosen architecture. Firstly, the choice of multiple acceptor agents instead of a single acceptor is because in case the single acceptor fails, progress is halted. A value is considered chosen when large enough set of acceptors have accepted it. Since it is required to choose a value even if only one value is proposed, an acceptor must accept the first proposal that it receives. Acceptor must be allowed to accept more than one proposal since there might not be a majority of acceptors choosing one value. Each value or also known as the proposal is tracked by assigning a number to each of them. If a proposal is chosen, then every higher numbered proposal issued by any proposer should have that chosen value. Whenever a proposal with value v and number n is issued, the majority of acceptors can be only in 2 states. Either no acceptor has accepted any proposal numbered less than n or v is the value of the highest numbered proposal among all proposals numbered less than n accepted by the majority of acceptors. The proposer must learn of the chosen value of the above rule is to stay intact. This is accomplished by the acceptor making a contract with the proposer to never accept any proposal that is lower than the proposed value. Therefore, before proposing a value, the proposer will send a prepare request that will create a promise never again to accept a proposal numbered less than n and send the proposal with the highest number less than n that it has accepted, if any. The learners get notified of the chosen value by receiving responses from acceptors regarding the accepted values. But the number of responses is equal to the product of the number of acceptors and the number of learners. To guarantee progress, a distinguished proposer must be selected as the only one to try issuing proposals. This is the overall flow of Paxos and has proven in many applications to be highly reliable and efficient.

2.14.2 Kafka / ZooKeeper

ZooKeeper [26] is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. ZooKeeper used the Zab [27] protocol which is a prominent member of the family of Paxos. The ZAB protocol ensures that the Zookeeper replication is done in order and is also responsible for the election of master/leader nodes and the restoration of any failed nodes. Underlying mechanism is similar to Paxos, Zab is a crash-recovery atomic broadcast

algorithm which is responsible for total order maintenance of messages and dependable delivery to ZooKeeper instances.

2.14.3 Raft

Raft [28] is the latest addition to protocols belonging to the family of Paxos. The outcome and the performance is similar to Paxos but the underlying structure is different to Paxos. One of the main key reasons to implement Raft is to increase understandability, decomposing the functions of leader election, message broadcast, log replication, view change and safety property. The key differences of Raft compared to other similar algorithms are:

- Strong leader election. Log entries flow from leader to other servers. It simplifies management of replicated logs and makes Raft easier to grasp.
- Raft uses randomized timers to elect leaders for each epoch. This feature will extend the heartbeat mechanism of similar consensus algorithms.
- A joint consensus approach is used for changing memberships for the servers in the cluster. It allows clusters to operate normally during configuration changes.

2.14.4 Quorum Chain

There are 3 types of nodes which participate in Quorum chain consensus [41]. The voter node is responsible for voting on the validity of the blocks. There are many voters at a particular time and the block with the highest number of votes passing the threshold value get appended to the blockchain. Block creation is allowed only by the maker node. They create the block and set their signature in the extra-data field of the block. The nodes which are neither maker or voter are called observer nodes. Observer does not take part in block making nor voting but instead will simply receive and validate blocks.

2.15 Byzantine Consensus

A byzantine node is a node which is subverted from its goal to achieve consensus in the network by an adversary and to act maliciously disrupting the normal process. In an environment where the amount of participant nodes is unknown and the access to the network is public (similar to public blockchain), the prominent protocols are proof-of-work, proof of stake, delegated proof of stake, proof of elapsed time and there are many others. In the case of a network where the nodes require a membership to access, it is possible to adhere to different family of protocols called practical byzantine fault tolerance (PBFT) [29]. It is an extension of Paxos/VSR and progresses through a sequence of epochs with an appointed leader. PBFT protocol tolerates one third of nodes acting maliciously and if this amount is exceeded, consensus is halted. Currently the only other variant of PBFT is BFT-SMaRT (<https://github.com/bft-smart/library>) which was actually developed before the interest in blockchains surged and have so far been experimented but seems to not yet used in production systems of permissioned blockchains [30].

2.15.1 Proof of Work

Proof-of-work is the first ever consensus algorithm that was adapted for permissionless blockchains introduced by Bitcoin [6]. In a network that adapts proof-of-work consensus, a set of nodes called miners compete each other to complete transactions, create a block and get rewarded. All participants of the network receive tokens which include transaction details. The miners need to validate these tokens and arrange them to a block. With this set of tokens, the miners start mining which is the process of solving a mathematical problem that can only be solved using brute force. Mainly idea is to create a hash with some specific property such as a hash with 5 zeros in the most significant digits of the hash value. The hash is created with values of the tokens, the hash number of the previous block, a nonce and some other minor values. When a miner succeeds in creating this hash value, his block gets appended to the chain and the block is broadcasted

to peer nodes. So far many cryptocurrency use proof-of-work as the consensus algorithm in their blockchain which include Bitcoin, Ethereum, Litecoin and many others.

2.15.2 Proof of Stake

Proof-of-stake [42] is a concept based on which a user can mine or validate a block according to how many cryptocurrency coins the user owns. This is designed as an alternative to proof-of-work which requires huge amount of computation energy for mining. While many features of proof-of-stake is similar to proof-of-work the main difference is that the mining power to the proportions of the coins held by the user instead of the computing power the user has.

2.15.3 Delegated Proof of Stake (DPoS)

An extension of proof-of-stake consensus algorithm. There are 2 steps in the process of achieving consensus in DPoS. First in electing a group of block creators whom are stakeholders of the network. The stakeholders are elected since they lose the most in case the network does not function as wanted. The next step is scheduled production where the block creator is allowed to create a block only during the provided time slot for that node.

2.15.4 Practical Byzantine Fault Tolerance (PBFT)

PBFT is an algorithm discovered in the year 1999 [29] more than a decade before blockchain was discovered. It received renewed attention with the popularity of blockchains. PBFT is mostly adapted in permissioned blockchains because the performance is too slow for widespread networks such as permissionless blockchains. The PBFT algorithm can be described as follows.

- A client sends a request to the primary server.
- The primary server duplicates this request and sends to its backup servers.
- The backup replicas execute the request and sent a reply back to the client.
- The client waits for 'f+1' requests from different replicas where f is the number of faulty nodes that can be tolerated by the network.

PBFT provides byzantine tolerance in asynchronous network environments such as the internet. Now there are many adaptations of PBFT such as BFT-SMaRT that are better performers.

2.15.5 BFT-SMaRT

BFT-SMaRT [43] is a byzantine tolerant consensus algorithm that are used in blockchains such as Symbiont and R3 Corda [2]. It implements a modular state machine replication protocol on top of PBFT. Here the client sends requests to all replicas which trigger the start of consensus. The leader of the consensus instance will propose a batch of requests to be decided by sending the batch to all replicas. The replicas validate whether the sender is the leader and whether the proposal is valid. If all is validated, each replica sends each other a message which include the cryptographic hash of the proposed batch. If a replica receives the required threshold of hash messages, the replica may accept the batch and sends accept message to all other replicas.

2.16 Overview of Blockchain Scalability

2.16.1 Permissionless Blockchains

As discussed in chapter 2.8, the consensus plane is one of the layers in blockchain that limits scalability. Let's consider consensus protocols that are used in permissionless blockchains today:

- Using the proof of work consensus, the transaction throughput of Bitcoin is around 7 trans/sec [18]. This value is achieved by setting 1MB block size and a 10-minute inter-block interval. This also means a client requires to wait an average of 10 minutes in order to verify that the transaction is included in the blockchain [20]. There is also the issue of energy efficiency where a large amount of CPU power is required to mine the blockchain. Bitcoin offers security and decentralization and thus lacks scalability according to the blockchain trilemma.
- The Etash consensus protocol used in Ethereum 1.0 [31] uses a different approach but similar to proof of work. Instead of dictating the block creator on raw computation performance, this algorithm is based on how fast data in a machine is moved around in memory. This is called "memory hardness". With this new protocol, Ethereum was able to achieve a throughput of 20 trans/sec. This is due to factors such as limiting the amount of gas that can be spent on each block [18]. Again the scalability has been prevented.

Although there are a number of other protocols that have been around, there is no evidence of these protocols being successful in production environments and there is no material or research on their performance and security when adapted to blockchain. However due to the nature of this research and the time frame to implement and test a consensus protocol for a permissionless blockchain is quite large, we will not focus on this area.

2.16.2 Permissioned Blockchains

Discussed in chapter 2.13, we can see that both crash tolerant and byzantine tolerant consensus are used in permissioned blockchains. But since permissioned blockchains allow only authorized nodes to participate in the network and the fact that many applications favor speed over security such as in the case in Litecoin [32], this research will be focused on improving blockchains with crash tolerance. We can further validate this scenario with the case of Hyperledger Fabric. Before Raft, the two consensus protocols that were natively supported by Fabric were Kafka and PBFT. In [33] we can see an evaluation carried out using PBFT. But now it is discontinued and the supported consensus models are Kafka and Raft. Even though neither support byzantine tolerance, it can be observed that the demand in having byzantine tolerance in permissioned blockchains is low. It is also the case that consensus that tolerate byzantine faults are considerably slower than consensus that tolerate only crash faults.

There are mainly two concepts of consensus approaches among the crash fault tolerant consensus. They are voting based or Paxos based. Protocols such as Kafka and Raft are Paxos based while protocols like QuorumChain are voting based. There are some drawbacks in the basic structure of these protocols [34].

- A single node is responsible for creating the set of transactions that will be appended to the chain. In the case of Paxos or Raft, the leader appointed in each epoch is responsible for collecting transactions from all its peer nodes, and broadcasts the ordered set of transactions back to its peers. In voting based, the block creator will collect all the votes until a vote threshold is reached and then propagate the block to peer nodes. It can be seen there is a central coordinator that is part of the main process of creating a block where this node carries the computation load in arriving to consensus. The scalability can be improved by distributing this task in some way among other nodes participating in the network.
- Another issue that arises of having a central coordinator is network traffic. If there is a large number of nodes in the network, all the transactions received by each node needs to reach the leader node. This can create large traffic in the network connecting the leader, which can bring down performance.

This research will address these issues and propose a consensus protocol that can overcome these bottlenecks and help scale blockchain and improve performance.

Table 2.2 - Comparison of features of several permissioned DLT platforms

Blockchain Platform	Primary Application	Source Code Language	Open Source / Proprietary
Hyperledger	Generic	Golang	Open Source
Corda	Financial	Java	Open Source
Ripple	Financial	C++	Open Source
Symbiont	Financial	Domain specific	Open Source
Tendermint	Generic	Golang	Open Source
Kadena	Business	?	Proprietary
MultiChain	Financial	C++	Open Source
HydraChain	Extension of Ethereum for permissioned blockchains	Python	Proprietary
Quorum	Financial	Solidity	Proprietary

2.17 Consensus Protocols Not Yet Adapted to Blockchains

[2] provides an analysis on several popular permissioned blockchains on their consensus. Among them Raft and BFT-SMaRT are the most popular for crash tolerant consensus and byzantine tolerant consensus respectively. Both protocols were adapted from the Paxos family of protocols. There are other consensus algorithms that have not yet been adapted to blockchains and could have the means to overcome the drawbacks discussed in chapter 2.15.2. Out of them, the Canopus consensus protocol seems to be the most promising.

2.18 Canopus

Canopus [34] is a parallel, network-aware, and decentralized consensus protocol designed for large-scale deployments. It has a few key differences compared to the traditional consensus algorithms used in permissioned blockchains. Paxos which is the widely used consensus protocol elect a centralized coordinator to enforce consensus on client request or proposal. The coordinator or leader is re-elected if it fails or crashes. Although many variants of Paxos have been developed, they have not improved the scalability of the protocol. Canopus does not broadcast read requests and manage total ordering by introducing minor delays to them. It is a network-aware protocol that organizes the nodes in the form of a wide-area overlay tree. Nodes located in the same rack form a virtual group, which can use any reliable broadcast or stronger protocol to reach agreement. Another way to increase performance is through exploiting network hardware. Canopus supports such optimizations that can improve throughput significantly.

There are a few design assumptions in the Canopus protocol. This research will not focus on these due to the fact of unavailable hardware resources and limited time frame. Some assumptions are rare full rack failures, rare network partitions between ToR switches, client coordinator communication is similar to other consensus algorithms where client sends request only to one node in the network and tolerance to crash stop failures only. Canopus detects node failures by using a method similar to the heartbeat mechanism in Raft.

One of the main reasons to select Canopus algorithm as the most suitable to improve the throughput of a blockchain network is its performance. The numbers show the improvement in performance is massive with over 3 million transactions per second for 20% write requests while over 1.5 million trans/sec for 50% write requests [34]. With this understanding it would seem that Canopus is a potential candidate for obtaining a scalable blockchain.

These are the key advantages of Canopus over Paxos based consensus protocols.

- Canopus is decentralized. Each node receiving transactions will order and broadcast to other node groups in the network. Does not rely on a central

coordinator which improves scalability when more nodes participate in the network.

- Super leaf groups and LOT overlay structure provide low network latency and prevent unnecessary traffic generated in the network by sending the ordered messages payload to a single node in the group and not broadcasting to all nodes in the network (Discussed further in chapter 3).
- High performance and scalable consensus protocol than Paxos.

2.19 Integrate Canopus to a Blockchain

In order to test the hypothesis that Canopus consensus protocol is able to scale blockchain performance, we will need to build a blockchain framework and integrate the consensus protocol. However, many permissioned blockchains support “pluggable consensus” which implies that the consensus algorithm can be changed easily and the blockchain framework supports any consensus protocol due to its modular architecture. Some frameworks that support pluggable consensus are R3 Corda and Hyperledger Fabric which are among the popular frameworks. For this research, Fabric will be the blockchain framework of choice due to its popularity and large developer community and support.

2.20 Hyperledger Fabric

Hyperledger project was initiated in 2015 by different companies interested in blockchain technology who wanted to pool their resources and create an open-source blockchain technology. According to [35], stakeholders of Hyperledger project believe that the future of blockchain will involve modular and open-source platforms that are easy to use. This fact is further confirmed by the current tendency of enterprises who choose open source platforms to reduce risks. Instead of reinventing the wheel, they develop industry-specific enhancements on top of a proven platform. Thus enterprises choose to “stand on the shoulders” of others who pioneered work and shared it with the world, rather than developing an entire infrastructure and engineer all of its solutions.

Hyperledger provides a greenhouse structure that incubates new ideas, supports a wide variety of frameworks and tools while consuming only a few essential resources.

Different scenarios where blockchain solutions have compelling use cases, have different requirements for confirmation times, decentralization, trust and other issues. Each issue represents a potential “optimization point” for the technology. Hence choice of the most suitable framework or tool for a given scenario is of key importance.

In order to address the diverse frameworks and tools, all Hyperledger projects follow the same design philosophy. Therefore, all Hyperledger projects are modular, highly secure, interoperable, cryptocurrency-agnostic and complete with APIs. Hyperledger frameworks are designed to be modular and extensible with reusable building blocks. Therefore, individual components could be changed without affecting the rest of the system. Based on different requirements, functional modules for communication, consensus, smart contracts etc. could be combined to build well-suited distributed ledger solutions. Interoperability with different blockchain networks is provided by ‘complete with APIs’ feature. According to [35], Hyperledger will never issue any cryptocurrency since their main aim is to facilitate development of generic applications. Out of the main business blockchain components of Hyperledger architecture mentioned in [36], obtaining an understanding of consensus layer is important for this research.

Hyperledger Fabric can be considered the most well-known blockchain platform in Hyperledger supported by IBM. This framework is chosen in order to test and evaluate the proposed consensus algorithm due to several supporting factors. Hyperledger Fabric is an open source project with a good community to provide support. The second contributing point is the ability to plug different consensus protocols to the ordering service of Fabric. Many other research projects too have evaluated other consensus algorithms using this platform.

2.21 Hyperledger Fabric Architecture

Hyperledger Fabric is one of the first ever extensible blockchain systems for running distributed applications. The support for modular consensus protocol was the attractive feature to choose this platform to test Canopus on blockchains. Another reason too would

be the large open source community that has accumulated throughout the years that can ease the development process. Fabric is developed using Golang, a programming language developed by Google. The source code can be downloaded through this link [37].

Currently Hyperledger Fabric has support for three consensus protocols natively. Table 2.1 provides details of these protocols. Each has its own usage and limitations. Comparing them with Canopus, the main components can be considered the leaf-only-tree overlay which will include the super leaves and at least one representative for each leaf. At least one node will be required to be running on each leaf or else the consensus process will be halted. Therefore, this provides crash fault tolerance to the ordering service. This will be highly scalable and can be used in production environments but it will be complicated to implement and deploy without knowledge of the Canopus consensus protocol.

There are three main phases in Hyperledger Fabric in order to complete the creation of a single block [13]. They are execute, order and validate phases.

2.21.1 Execution Phase

A smart contract in Hyperledger Fabric is called chaincode, which is a program code that implements the application logic and runs during the execution phase. Clients would sign and send a transaction proposal to one or more endorsers for execution. Endorsers are set through an endorsement policy when deploying the blockchain. An endorsers role is to simulate the transaction proposal received by executing the operation on the specified chaincode which has been installed on the blockchain. The chaincode runs in a Docker container, isolated from the main endorser process. As the result of the simulation, this will generate a read and write set consisting of the state updates that would go through if the transaction is validated and committed. This set is sent back to the client in a proposal response. When the client has collected enough endorsements on a proposal, then it submits to the ordering service.

Table 2.3 Comparison of pluggable ordering services for Hyperledger Fabric

	Solo	Kafka	Raft
Components	Single ordering service, no replication.	Decentralized, replicated ordering service. Contains Apache Kafka Cluster.	Decentralized, replicated ordering service. Ordering cluster contain minimum of 3 nodes.
Usage	For testing only	For production where nodes are trustworthy	For production where nodes are trustworthy
Advantages	Easy to deploy Require less resources	Reliable Scalable	Tolerance for CFT.
Disadvantages	Single point of failure. Not scalable.	No Byzantine fault tolerance.	Not scalable Complex
Fault tolerance	None	CFT	CFT

2.21.2 Ordering Phase

The client can send the set of endorsements including the transaction payload to any orderer node that is deployed. When the orderer node receives a certain amount of transactions or reaches a certain timeout, the orderer nodes will establish a total order on all submitted transactions per channel. After arriving at consensus over the order of transactions, the ordering service batches multiple transactions into blocks and outputs a hash-chained sequence of blocks containing transactions. Grouping or batching transactions into blocks improves the throughput of the broadcast protocol, which is a well-known technique used in fault-tolerant broadcasts.

2.21.3 Validation Phase

There are 3 main sequential steps in the validation phase when a block is received from the ordering service.

- The **endorsement policy** evaluation occurs in parallel for all transactions within the block. If the endorsement is not satisfied, the transaction will be marked as invalid and its effects are disregarded.
- A **read-write conflict check** is done for all transactions in the block sequentially.
- Finally, the **ledger update phase** is executed in which the block is appended to the locally stored ledger and the blockchain state is updated. After the update, the client will be notified whether the transaction has been committed or disregarded.

Chapter 3 - Design

3.1 Introduction

This chapter will present the details of the proposed consensus algorithm, Canopus. It includes a detailed description of how Canopus algorithm functions, what are the alterations made to the original algorithm presented by the paper [34] and how it will fit inside the Hyperledger Fabric framework. The overall objective of what this design will accomplish is also stated at the end of this chapter.

3.2 What Canopus Is & Not

Distinguishing the features of Canopus is important compared to other consensus protocols to understand how this algorithm is more refined and optimized and has the potential to be the next best consensus algorithm for permissioned blockchains. The following sections will discuss some main features of Canopus while also discussing any limitations to implement these features in this research.

3.3 Centralization & Decentralization

Paxos [24] was the most successful and established consensus protocol that was widely used in many distributed systems. It's atomic broadcast capability was highly reliable and had high throughput numbers. One main feature of this algorithm is the central coordinator as discussed in chapter 2. This is also the case in Raft consensus. The central node bears the weight of all the processing happening in the network for consensus and has to communicate with its peer nodes to retrieve the approval for the proposed ordered set of transactions. This is a limitation to scalability where when more nodes participate in the network, the more time it takes to send and retrieve messages among them.

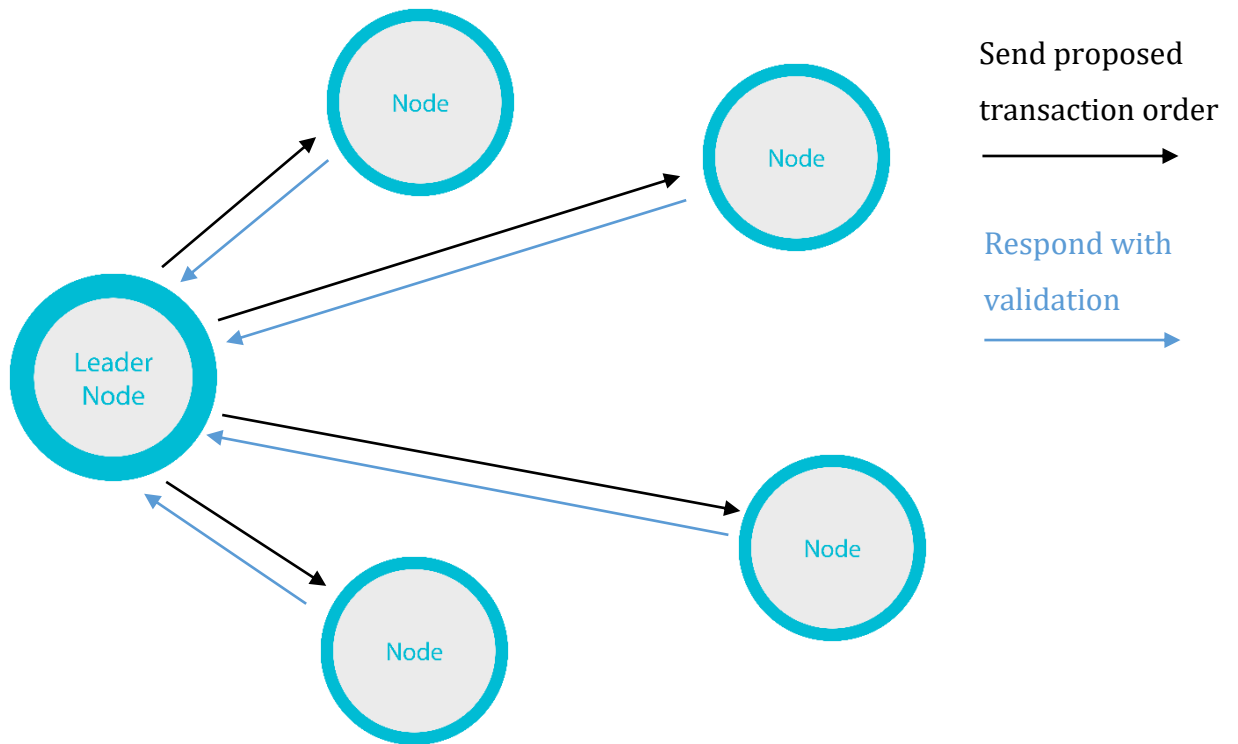


Figure 3.1 - Illustration of Raft consensus

Canopus eliminates this barrier of having a central coordinator. All nodes in the network participating in consensus will be allowed to receive transactions from peers. Each node will order these transactions to the time of arrival. When the consensus cycle starts, these transactions are distributed methodically where then each Canopus node will receive a subset of ordered nodes. They are again ordered on the large random number generated by each node that is also shared with the transaction payload. There are representative nodes that play an important role in each super leaf node. The representative node is responsible of retrieving transaction payloads from other branches of the leaf-only-tree network topology. This will be discussed in the next chapters. Overall the algorithm removes the central coordinator barrier that will allow to distributing the processing payload to all nodes in the network.

3.4 Exploiting Network Hardware

There are several consensus algorithms that take advantage of high end network hardware to achieve more performance. Devices like low latency switches, high performance machines, fast data communication mediums (fiber cables) and other specialized hardware. Canopus too can take advantage of such hardware. But due to unavailability of such resources, the performance gains from hardware exploits won't be tested in this research.

3.5 Leaf Only Trees

Many of the current consensus algorithms broadcast the messages to its peer nodes in the network. But in Canopus this is not the case. Each Canopus node need to disseminate the write requests it receives to every other node in a series of rounds. The most common way to approach this is to broadcast the requests to every node group which could flood the network quickly depending on the incoming traffic of requests from peers. But instead Canopus rely on message dissemination which follows paths on a topology aware virtual tree overlay. The overlay used here is the Leaf Only Tree overlay(LOT) [44]. There are a few distinguishing features of LOT.

- There are 2 types of nodes in a LOT, **physical nodes and virtual nodes**. Physical nodes as the name implies are the only real dedicated process that is running in the network as a service. Internal nodes in the tree are all virtual nodes in the case they do not exist as a separate service running in the network. But instead these virtual nodes are mapped to physical nodes when nodes in other branches in the network attempt to communicate with such a node. Meaning each physical node emulates all of its ancestor nodes.
- Each physical node emulates its ancestor virtual nodes. Therefore, each physical node becomes aware of the state of each of its ancestor nodes. This allows any of the physical nodes to respond to query requests when any of the virtual nodes are queried. The replication is also an advantage in a case of a node failure where

another node having replicated the same set of ancestor nodes can respond to a query.

- All the physical nodes in a server rack or machine will be grouped in to a single super leaf. There are mainly 2 reasons for this design. Firstly, this reduces the number of messages that are exchanged between any 2 super leaves. Instead of having all to all communication between every physical node in the super leaves, a subset of super leaf nodes called representatives of the super leaf will communicate with the representative of the another super leaf. Second, because all the physical nodes in a super-leaf replicate their common parents' state, a majority of the super-leaf members need to simultaneously fail to cause the super-leaf to fail.

For this research, the blockchain platform where the consensus algorithm is tested won't be deployed in multiple datacenters due to cost and unavailability but will be tested over virtual machines hosted in the cloud using Docker to deploy. Here then we assume the latency that will occur over the network is insignificant.

3.6 Consensus Cycle

The protocol divides execution into a sequence of consensus cycles. Ordering of the write requests is done by having each node, for each cycle, independently choose a large random number, then ordering write requests based on those random numbers. Requests received by the same node are ordered by their order of arrival. In each cycle a node disseminates the write requests it receives during the previous cycle to every other node. Message dissemination follows paths on a topology aware virtual tree overlay called the Leaf-Only Tree (LOT) overlay [44]. Three distinguishing properties of LOTs are presence of physical and virtual nodes, node emulation where physical nodes emulate its ancestor nodes and super leaves are nodes located in the same rack.

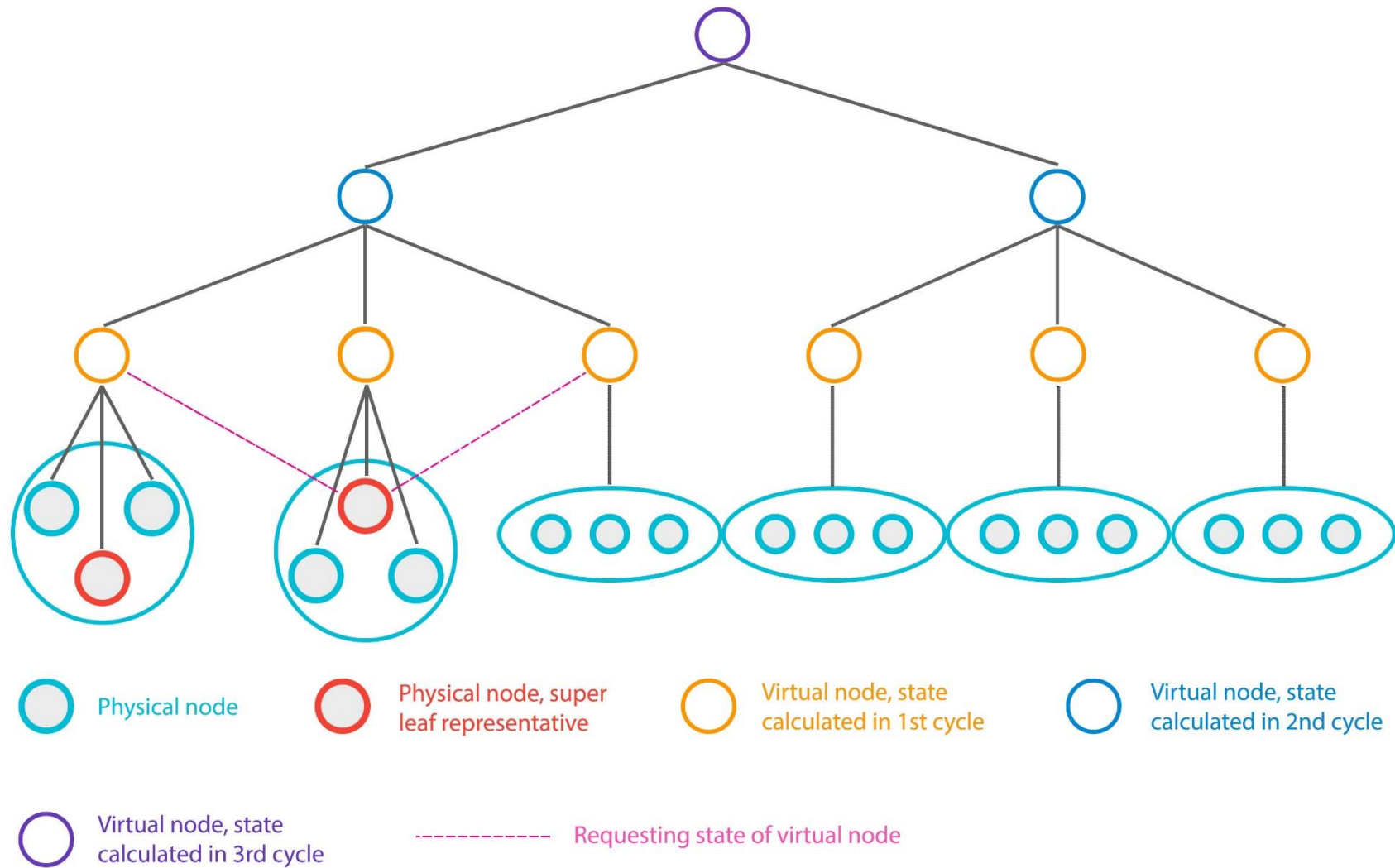


Figure 3.2 - LOT in Canopus

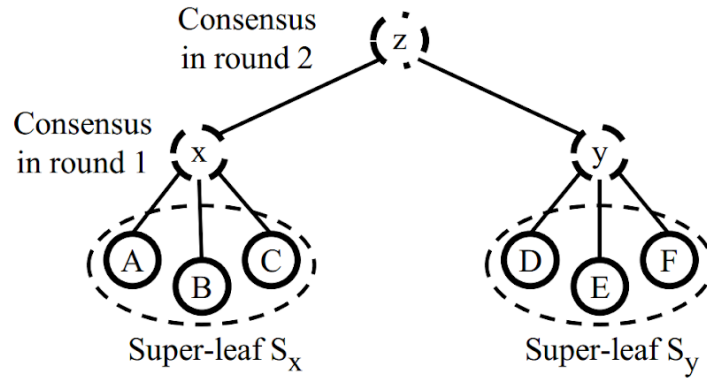


Figure 3.3 - Six nodes arranged in two super-leaves in a LOT of height 2

The first round of the consensus cycle will:

- Prepare a proposal message and broadcast the message to the peers in its super-leaf using a reliable broadcast protocol.
- Each node generates a random large number called a proposal number.
- Each physical node independently orders the proposals according to their proposal numbers.

Other rounds in the consensus cycle:

- Representative node sends a proposal-request to emulators of the virtual node in order to retrieve that virtual nodes state.
- Receiver node replies with the proposal message belonging to the required virtual node.
- After receiving all proposals, compute the order of all collected write requests.
- Repeat until the state of the root virtual node is computed.

According to figure 3.3, there are two super leaves each having 3 nodes. At the end of the first consensus cycle, all nodes in super leaf S_x will have calculated the state of x while all the nodes in super leaf S_y would have calculated the state of y . Then a representative from each super leaf will share the state of their immediate ancestor to start the second consensus cycle. This will be the last cycle in which all nodes would have calculated the state of z which is the root and ancestor of all nodes in the LOT.

Figure 3.4 shows the high level architecture of the design when Canopus is plugged into Hyperledger Fabric framework. Clients send requests to peer nodes, peers endorse and

send a response back to clients. Clients then send the endorsements to orderer nodes. Orderer nodes arrive at consensus with regard to all endorsement received by clients. The ordered transactions are sent to peers for validation. The peers finally respond to clients whether the requests were successfully processed or not.

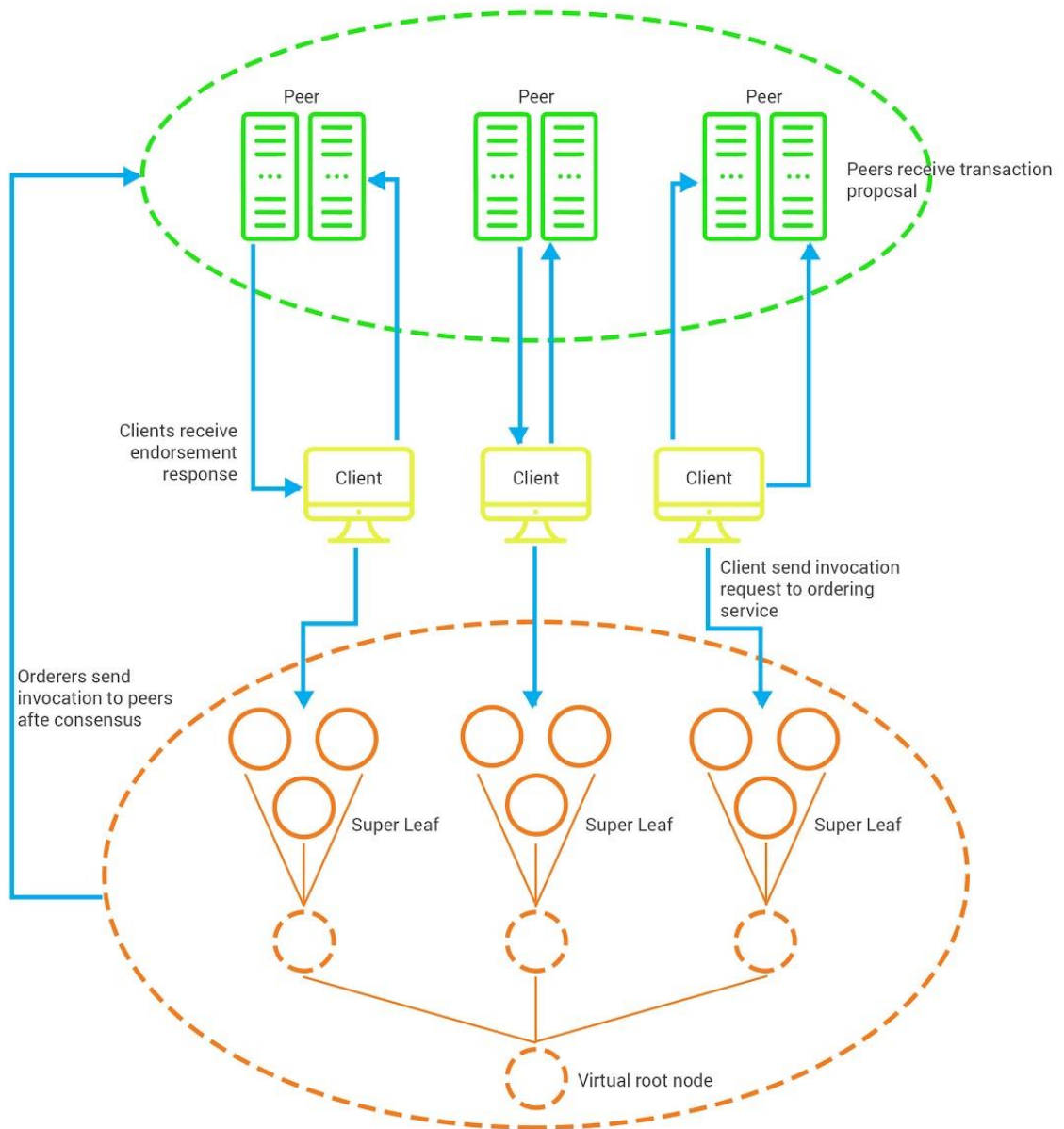


Figure 3.4 - High level architecture of proposed blockchain network

3.7 Conclusion

This chapter presented the overall architecture and processes that are involved in the design process of Canopus algorithm. With the understanding of the blockchain process in Hyperledger Fabric, it becomes much clearer how the orderer service fits in and how the Canopus algorithm will be implemented. Next chapter presents the details of implementation of Canopus in Hyperledger Fabric, the interfaces Hyperledger has to implement a consensus, the structure of the source code and how to deploy the newly built algorithm as an ordering service.

Chapter 4 - Implementation

4.1 Introduction

This chapter will provide the most relevant implementation details with regarding to developing the Canopus algorithm within the Hyperledger Fabric framework. Appendix B will contain further code listings that went into the source code of this implementation. Sections will follow through each step of deploying a Fabric network and what changes were required to make in order to support Canopus consensus algorithm. For simplicity of explaining the implementation process, we will assume the development of a LOT which contain 5 nodes and 2 super leaves each having 3 and 2 nodes respectively.

4.2 Define Organizations

Each peer and orderer node in the Fabric network should belong to an organization. The organizations are a virtual construct of where the peers and orderer nodes reside in the network. An organization can have either peer nodes or orderer nodes and not both. The organizations of the network are defined in a *configtx.yaml* file. This file is used when generating the genesis block and the channel configuration for the network. Each organization defined require a name, ID, the directory which include the membership service provider configurations and the policies. When using TLS which we will when implementing Canopus, each organization in the network requires authorized to verify its identity. Policies are restrictions that are enacted on the network for each type of node. There are mainly 4 types of user nodes and when defining an organization, it is required to define the policies for each of these user types.

- Readers
- Writers
- Admins
- Endorsement (only in peer organizations)

Peer organizations are required to define the location of anchor peers which are used for cross gossip communication.

In the case of deploying a Canopus consensus protocol, it is possible to have all the physical nodes of the network to belong in a single organization or have a separate organization for each super leaf. If a different organization requires control of some subset of the orderer nodes, a separate organization for the subset of super leaves is the way to go when deploying. Else having one organization is sufficient. Refer appendix C.1 for code segment.

4.3 Generate Cryptographic Certificates

All nodes and clients including the Canopus nodes will require cryptographic certificates to communicate using TLS. Hyperledger Fabric has an inbuilt tool to generate these certificates. When using the blockchain in a production environment, it is possible to use any other certificate authority than the one provided by Fabric. The *crypto-config.yaml* file is used to define the nodes and their organizations in order to generate the certificates. This file is consumed by the *cryptogen* tool and the certificates are separated through directories (appendix C.2).

4.4 Define Orderer Related Parameters & Profile

There are several parameters that can be adjusted on the orderer service before deployment. Some of these parameters are:

- **BatchTimeout** – The amount of time to wait after receiving a single transaction payload by an ordering node on a channel before creating a batch to generate a block. In the case of Canopus, when this timeout is reached, we start the consensus cycles by first sending the current payload to other nodes in its super leaf and requesting the payload from relevant virtual nodes. A timeout is important in a scenario where the transactions in the network are relatively less and it takes a long time to reach the batch message limit.

- **BatchSize** – Defines 3 sub parameters related to the size requirement of each block.
 - **MaxMessageCount** – The maximum number of messages that is allowed to a block is defined here. A batch will be created as soon as this limit is reached regardless of the timeout.
 - **AbsoluteMaxBytes** – The maximum size of the block is defined here. Regardless of message count and timeout, if the current transaction payload exceeds this value, a batch will be created.
 - **PreferredMaxBytes** – This parameter has the least priority. If at one moment the payload has reached this limit, it will attempt to create a batch.
- **Addresses** – The addresses of all the orderer nodes need to be included here.

Orderer settings are defined in *configtx.yaml* file. The default settings for each consensus algorithm is included under the *Orderer* field. It is possible to define profiles that override these values. This increase the easiness of creating new orderer nodes and changing their configuration. In case no profile flag is provided, the default values will be written to the genesis block (refer appendix C.3 for relevant code snippets).

4.5 Generate Channel Artifacts

After properly setting up the configuration files, we generate artifacts that is required by the Fabric blockchain to initialize and configure. There are mainly 2 artifacts that are required for this purpose.

4.5.1 Genesis Block

The genesis block is a configuration block that initializes the ordering service, or serves as the first block on a chain. It includes all the configuration details defined in *configtx.yaml* file. When generating the genesis block, in order to generate the orderer nodes for Canopus, we can provide the name of the profile where the Canopus consensus

nodes are defined. Then the specific parameters get written to the genesis block. When bootstrapping the ordering service, the values written to the genesis block is been used. Therefore, creating the genesis block is a vital step in the process of deploying the right consensus algorithm in the network.

4.5.2 Channel Configuration Transaction

In order to initialize or change the configuration of a channel over the Fabric network, we need to issue a configuration transaction. These transactions too get sent over the orderer service and then committed into blocks. For initialization configuration, we can create the transaction using the *configtxgen* tool. This transaction is then sent to the network after it is deployed. When creating this transaction, we provide a channel name, any profile of channel configuration and the output directory.

```
muraad@Root-5533:~/Documents/Projects/Research/fabric-samples/first-networks$ ./bin/configtxgen -profile CanopusConsensus -channelID byfn-sys-channel -outputBlock ./channel-artifacts/genesis.block
2020-01-05 09:15:22.630 +0530 [common.tools.configtxgen] main -> INFO 001 Loading configuration
2020-01-05 09:15:22.704 +0530 [common.tools.configtxgen.localconfig] completeInitialization -> INFO 002 orderer type: etcdraft
2020-01-05 09:15:22.704 +0530 [common.tools.configtxgen.localconfig] completeInitialization -> INFO 003 Orderer.EtcdRaft.Options unset, setting to tick_interval:"500ms" election_tick:10 heartbeat_tick:1 max_inflight_blocks:5 snapshot_interval_size:20971520
2020-01-05 09:15:22.704 +0530 [common.tools.configtxgen.localconfig] Load -> INFO 004 Loaded configuration: /home/muraad/Documents/Projects/Research/fabric-samples/first-network/configtx.yaml
2020-01-05 09:15:22.782 +0530 [common.tools.configtxgen.localconfig] completeInitialization -> INFO 005 orderer type: solo
2020-01-05 09:15:22.782 +0530 [common.tools.configtxgen.localconfig] LoadTopLevel -> INFO 006 Loaded configuration: /home/muraad/Documents/Projects/Research/fabric-samples/first-network/configtx.yaml
2020-01-05 09:15:22.784 +0530 [common.tools.configtxgen] doOutputBlock -> INFO 007 Generating genesis block
2020-01-05 09:15:22.785 +0530 [common.tools.configtxgen] doOutputBlock -> INFO 008 Writing genesis block
muraad@Root-5533:~/Documents/Projects/Research/fabric-samples/first-networks$
```

Figure 4.1 - Command line to build genesis block

Next step is to start up the network. Before this step of course, we require to implement the consensus algorithm and create a Docker image in order to deploy the algorithm. Next section will provide in depth details of the implementation process.

4.6 Implementing Canopus Consensus

4.6.1 Introduction

This section will provide implementation details of each step of achieving the Canopus consensus. Details from when an ordering node receives messages from peers and nodes to finally writing blocks to the chain and committing the transactions over to peers is presented here.

4.6.2 Receiving Transaction Messages

There are 2 ways an orderer node receive messages. The first is from a client whom have received the sufficient endorsements from peer nodes. The payload from the client will arrive as a single message. Hyperledger Fabric have pre-defined interfaces to handle messages arriving from clients. This interface has to be adapted by all consensus algorithms implemented in Fabric. The interface that defines a way to inject messages for ordering is **Chain**. The **Order** function in this interface accepts the messages from clients while the **Configure** function accepts messages that are related to channel configuration transactions. Due to time limitations, we will not be working on configuration transactions. The orderer node receive an **Envelope** object which include the payload and other metadata.

The second route of receiving messages is from another orderer node. When starting the consensus cycle, a orderer node will receive transactions from other nodes in the network. We implement a dispatcher that will manage the retrieval of such messages. The **OnSubmit** method will provide the ID of the orderer node who sent the payload and the payload itself. A difference to messages received by this way is the payload will include the random number generated by that orderer node for that specific cycle that is required to order the messages before committing.

When messages arrive from either way, the **Submit** function is called passing in the retrieved message with the sender ID.

```

in.go • dispatcher.go ×
er > consensus > etcdraft > dispatcher.go > ...
type ReceiverGetter interface {
    // ReceiverByChain returns the MessageReceiver if it exists, or nil if it does
    ReceiverByChain(channelID string) MessageReceiver
}

// Dispatcher dispatches Submit and Step requests to the designated per chain in
yacovm, a year ago | 1 author (yacovm)
type Dispatcher struct {
    Logger          *flogging.FabricLogger
    ChainSelector ReceiverGetter
}

// OnConsensus notifies the Dispatcher for a reception of a StepRequest from a g
func (d *Dispatcher) OnConsensus(channel string, sender uint64, request *orderer
receiver := d.ChainSelector.ReceiverByChain(channel)
if receiver == nil {
    d.Logger.Warningf("An attempt to send a consensus request to a non existing
return errors.Errorf("channel %s doesn't exist", channel)
}
return receiver.Consensus(request, sender)
}

// OnSubmit notifies the Dispatcher for a reception of a SubmitRequest from a gi
func (d *Dispatcher) OnSubmit(channel string, sender uint64, request *orderer.Su
fmt.Println("Dispatcher onsubmit : ", channel, " : ", sender)
receiver := d.ChainSelector.ReceiverByChain(channel)
if receiver == nil {
    d.Logger.Warningf("An attempt to submit a transaction to a non existing chan
return errors.Errorf("channel %s doesn't exist", channel)
}
return receiver.Submit(request, sender, true)
}

```

Figure 4.2 - Dispatcher.go functions to receive messages from other orderer nodes

4.6.3 Queue Messages and Trigger the Start of Consensus

The messages received by an orderer node from clients can be ordered to the time it arrives. One way is to set a timestamp for each message that arrives and sort them before creating a batch. This is an extra step and the computation time increases unless they can be maintained in some sort of data structure that support well with asynchronous access to that object. Golang channels can do just this. In Go a channel is a medium through which a thread can communicate with another thread which is also lock free. Threads in go are called goroutines. They are light weighted and easy to implement. With channels we can maintain not only the messages received through the **Order** function but

messages received from the **Dispatcher**. We set up the channels to the Chain object as shown in figure 4.3.

```
141 type Chain struct {
142
143     // for Canopus
144     lock          *WriteLock
145     currentNode   int
146     ordererC      *[]chan *orderer.SubmitRequest
147     maxMsgCount   int
148     receivedFrom  chan uint64
149     pendingBatches chan [][]*common.Envelope
150
151     configurator Configurator
152
153     rpc RPC
154
155     raftID      uint64
156     channelID   string
157
158     lastKnownLeader uint64
159
160     submitC chan *submit
161     applyC  chan apply
162     observeC chan<- raft.SoftState // Notifies external observer on leader change (passed in option)
163     haltC    chan struct{} // Signals to goroutines that the chain is halting
164     doneC    chan struct{} // Closes when the chain halts
165     startC   chan struct{} // Closes when the node is started
166     snapC    chan *raftpb.Snapshot // Signal to catch up with snapshot
167     gcC      chan *gc // Signal to take snapshot
168
169     errorCLock sync.RWMutex
170     errorC     chan struct{} // returned by Errored()
171
172     raftMetadataLock sync.RWMutex
173     confChangeInProgress *raftpb.ConfChange
174     justElected        bool // this is true when node has just been elected
175     configInflight       bool // this is true when there is config block or ConfChange in flight
176     blockInflight        int // number of in flight blocks
177 }
```

Figure 4.3 - Chain struct in chain.go file

The amount of elements in the channel can be easily obtained through the *len* function. When the number of messages in the channel exceeds the batch max message count which was provided in *configtx.yaml* file, the function to start the first cycle of consensus is triggered. This is also triggered in the case where the time after receiving the first message exceeds the batch time out.

There is also a third scenario where the consensus cycle should be triggered. Consider when an orderer node has not received any transactions. It would be staying idle till it receives any transactions. The idea behind receiving a message through dispatcher means another node is ready to start the consensus cycle. Here too it will be necessary to trigger the start of consensus cycle of that orderer node.

4.6.4 Distribute Messages

When distributing messages among the orderer nodes, the data format in which the message is wrapped should be able to encode and decode quickly for better performance. Most common data formats that allow to serialize and transmit are JSON and XML. Here we use Protocol Buffers or known as **Protobufs**, a data structure introduced by Google. Due to its efficient serialization and deserialization, it is most suited for a network with large traffic and interchanges many messages. In order to send the messages received by an orderer node to other nodes, we define a *rpc* object file that submits these transactions to other nodes in the network given the recipients node ID. The proto file and the relevant code is shown in figure 4.4.

```
95 // SendSubmit sends a SubmitRequest to the given destination node.
96 func (s *RPC) SendSubmit(destination uint64, request *orderer.SubmitRequest) error {
97     if s.Logger.IsEnabledFor(zapcore.DebugLevel) {
98         defer s.submitSent(time.Now(), destination, request)
99     }
100
101     stream, err := s.getOrCreateStream(destination, SubmitOperation)
102     if err != nil {
103         return err
104     }
105
106     req := &orderer.StepRequest{
107         Payload: &orderer.StepRequest_SubmitRequest{
108             SubmitRequest: request,
109         },
110     }
111
112     s.submitLock.Lock()
113     defer s.submitLock.Unlock()
114
115     err = stream.Send(req)
116     if err != nil {
117         s.unMapStream(destination, SubmitOperation)
118     }
119     return err
120 }
```

Figure 4.4 - SendSubmit function to distribute messages with other nodes in the orderer service

4.6.5 Calculate The Final State

The state of each virtual node gets calculated when the messages from the relevant nodes are received. Considering the example scenario, orderer 1 waits until orderer 2 and orderer 3 sends their transactions. When they are received, the state of the parent node can be calculated. Then this total order of transactions is sent to a representative of the

other super leaf. To calculate the final state, orderer 1 waits for messages from orderer 4 or orderer 5. The message received from either should be again distributed among the super leaf nodes of that orderer. Therefore, it can be clearly seen depending on the number of orderer nodes, the number of super leaves, whether the orderer node is a representative and the structure of LOT, the functions of each orderer differs from another.

When the final state is calculated, the messages are passed sequentially to the **Ordered** function of the **Blockcutter** object. The **Blockcutter** creates batches that are then used to create a **Block** object. This block is finally passed to the **writeBlock** function that writes the block to the chain and sends to the peer for validation. These functions are built in the Fabric source code and can be reused for Canopus consensus too (refer appendix C.5).

4.7 Build Orderer Package & Docker Image

The **Makefile** file in the root of the Hyperledger Fabric source code provide make commands to easily build the orderer Go package and the Docker image. We simply run these commands when changes on the files are completed. Refer appendix C.4 for commands and outputs in terminal.

4.8 Start and Initialize Fabric Network

Starting the network include creating and starting the containers for each peer and orderer node in the Fabric network. This is accomplished by writing a Docker compose file that defines each container and the required parameters for each container. Mainly this would include defining the services which are peers and orderers, container name, image name, port and the volumes where the binaries of Fabric and certificates needs to be copied.

Finally, we invoke configuration transactions to initiate the channel, join the peers to the channel and install the chaincode.

4.9 Summary

The chapter provided all required details of the implementation of Canopus algorithm in Hyperledger Fabric and deploying the blockchain. The Hyperledger Fabric version used for development was **1.4.4**. The process of creating a new consensus is shown and the implementation of the Canopus algorithm is presented in section 4.6. The steps to deploy the blockchain network and changes required to make in order to facilitate another algorithm is also presented.

Chapter 5 - Results and Evaluation

5.1 Introduction

The main purpose of the evaluation is to provide answers to the fourth research question. This chapter provides details regarding the benchmark results of Hyperledger Fabric running Canopus algorithm and compared to its natively supported consensus protocols, Kafka and Raft. Evaluation model followed in this chapter was based on [47] while sub evaluations were performed to verify speculations made during main evaluation process.

The implementation provided in chapter 4 and the evaluation process was performed using Hyperledger Fabric version 1.4.4. The benchmarking of each instance of different ordering nodes and consensus protocols were performed on an AWS t2.large instance (2 vCPUs, 8GB RAM, 64bit), Docker version 18.09.2 and Docker compose version 1.23.2.

5.2 Evaluation Process

The evaluation process will determine 2 main metric values discussed in chapter 2.4 which are transaction throughput and latency for Hyperledger Fabric running Canopus protocol. These metrics were compared to the performance of Kafka and Raft which is natively supported by Fabric. Each consensus protocol is evaluated against increasing number of ordering nodes and their throughput and latency values were observed. The details regarding the configuration of Fabric is listed below.

- The network included 2 peers each belonging to a different organization. The peers are connected to a single channel with the ordering service and the chaincode is installed on this channel. These 2 peers act as the endorsing peers for each transaction submitted by client nodes.
- 2 client nodes submitted transactions to the network with a fixed rate of 100 transactions per second. This value is chosen because neither consensus protocol regardless of different configurations ever reached this value.

- There are 2 smart contracts implemented in the chaincode for testing.
 - Open-account smart contract which simulates opening new accounts in the ledger with a provided value as the account balance. This function is independent of the rest of the data in the ledger of the blockchain.
 - Transfer-money smart contract which simulates money transfer between 2 randomly selected accounts for provided value. Since this is dependent of the state of the accounts concerned, generally the throughput value of this contract is much lower than the open-account smart contract.
- The batch time out parameter is set to 2 seconds. The maximum message count for each block is limited to 50 and the maximum block size is set to 1 megabyte.
- 4 Kafka instances and 2 ZooKeeper instances were deployed when evaluating the performance of Kafka consensus.

5.3 Evaluation of Consensus Protocols

As pointed out in [19], the consensus plane plays an important role in the scalability and performance of blockchains. Here we evaluate a decentralized consensus protocol not yet adapted to permissioned blockchains and compare to the most prominent consensus mechanisms Paxos and Raft. The evaluations were under taken using a tool called Hyperledger Caliper [46] which is a benchmark tool for all Hyperledger blockchain products.

5.3.1 Evaluation of open-account smart contract

The transaction throughput and latency were evaluated on the open-account smart contract against 3 consensus algorithms, Kafka, Raft and Canopus.

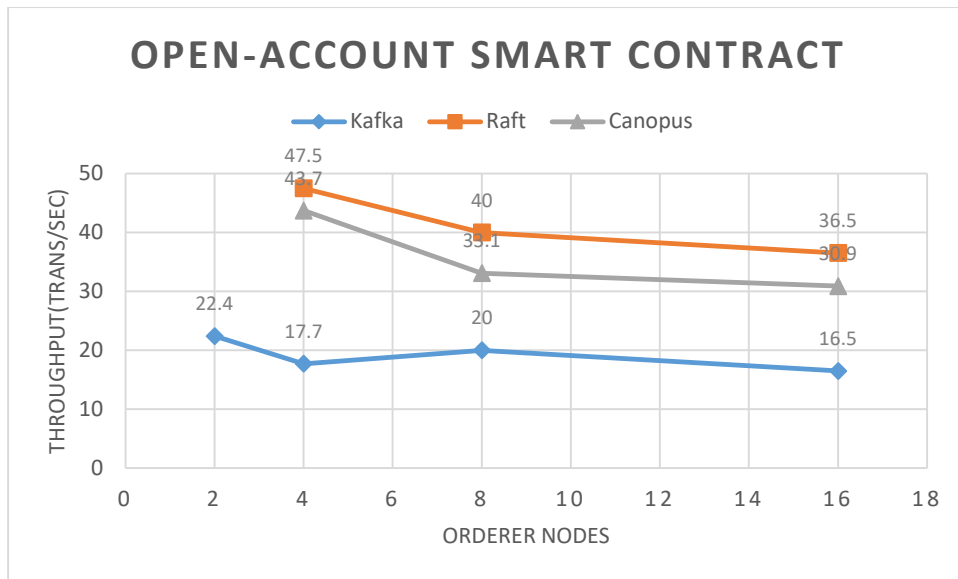


Figure 5.1 Transaction throughput of consensus algorithms on open-account smart contract

It can be observed from figure 5.1 that the highest performance is seen from Raft consensus, followed by Canopus and at last Kafka. The throughput decreases with increasing number of orderer nodes as expected. It is clearly observed that Kafka performs weaker than Raft and Canopus. It uses the Paxos protocol which is quite outdated comparatively. Even though we expected Canopus to perform comparatively better than Raft, this is not the case here. It has a closer performance to Raft but throughput deprecates in the same gradient as Raft with increasing number of orderer nodes.

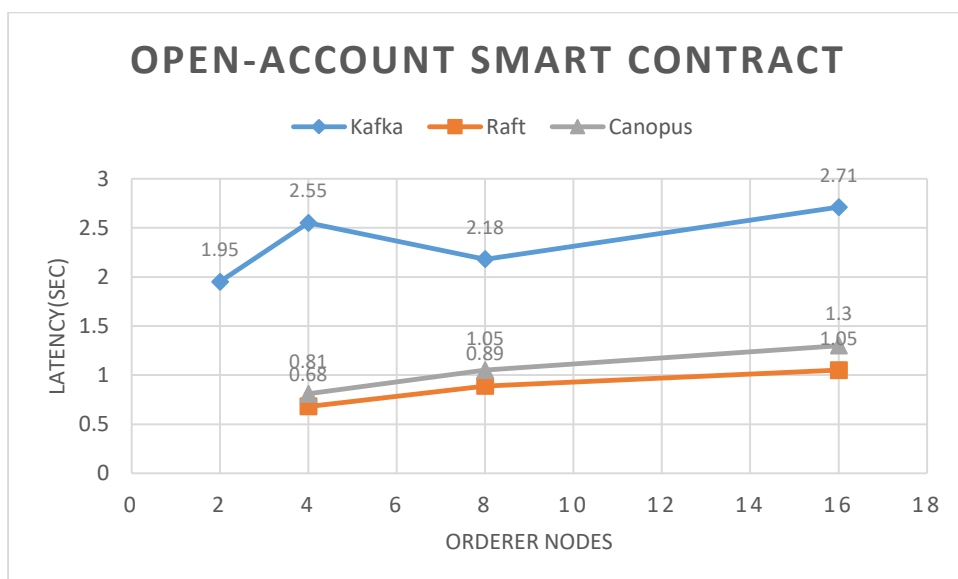


Figure 5.2 Latency of consensus algorithms on open-account smart contract

The same evaluation can be said from figure 5.2 with regard to latencies of the 3 consensus protocols. Kafka which is the slower among the 3 has the highest latency while again the latencies of Canopus and Raft are very similar with Raft reporting the lowest latency values.

5.3.2 Evaluation of transfer-money smart contract

The transaction throughput and latency were evaluated on the transfer-money smart contract against 3 consensus algorithms, Kafka, Raft and Canopus.

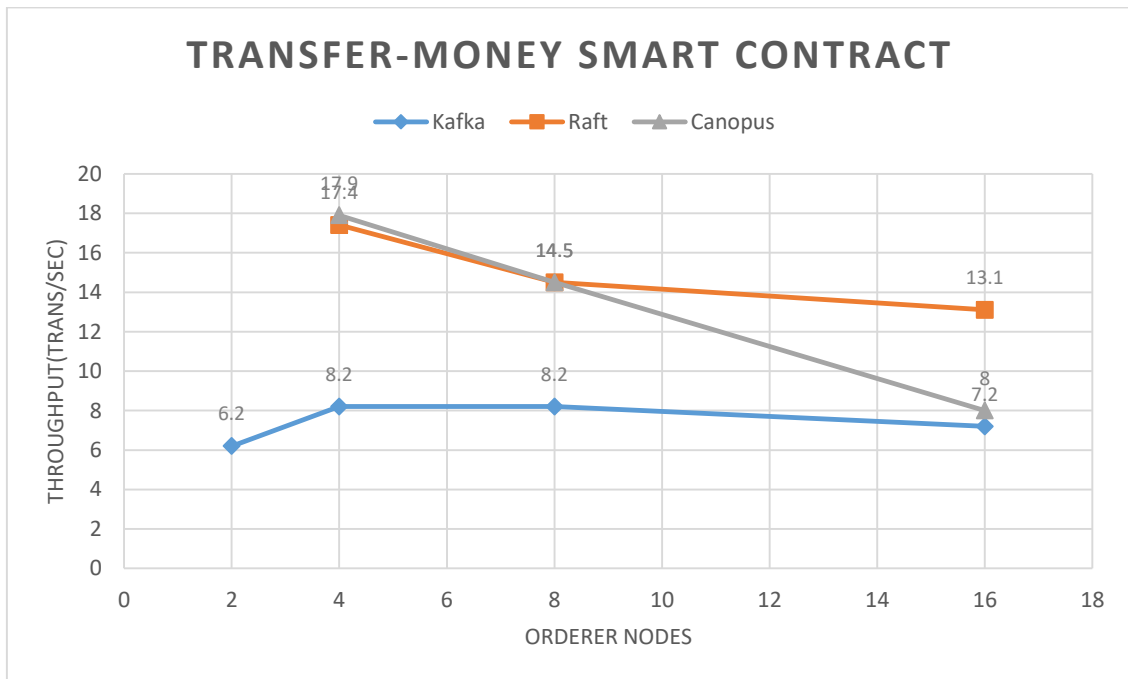


Figure 5.3 Transaction throughput of consensus algorithms on transfer-money smart contract

Compared to open-account smart contract which had a maximum throughput of 47.5 trans/sec, the maximum of transfer-money smart contract is around 18 trans/sec as observed in figure 5.1 and figure 5.3. One main reason for this is since the contract depends on the accounts which transfers occur; some transactions get invalidated when 2 or more transactions occur on the same account. Therefore, the portion of transactions that get invalidated do not make to the block and not considered as a processed transaction.

The same trend can be seen here as the open-account smart contract shown in figure 5.3. Kafka performs the weakest among the 3. Canopus performs slightly better than Raft up to around 8 orderer nodes but drastically drop afterwards. The throughput overall decreases with increasing number of orderer nodes on all 3 consensus algorithms.

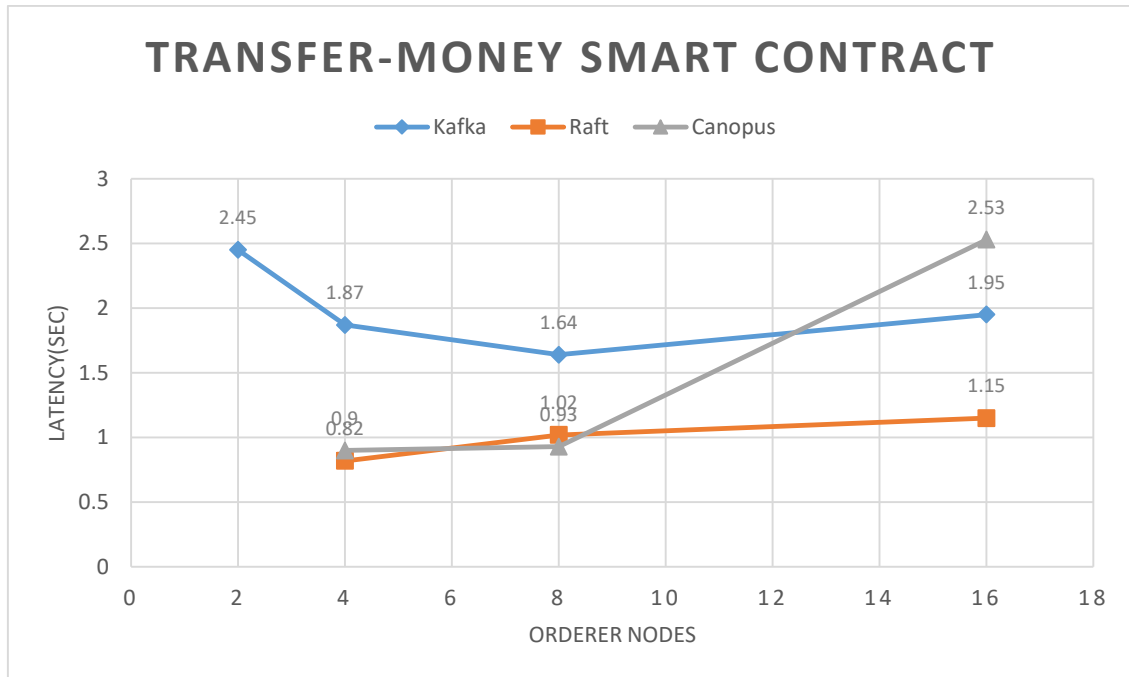


Figure 5.4 Latency of consensus algorithms on transfer-money smart contract

The same pattern can be seen in latency values of the 3 algorithms from figure 5.4.

5.4 Summary

Through the evaluation performed in this chapter, it can be inferred that Raft consensus protocol performs the best among the other protocols. By the hypothesis, we were expecting Canopus due to its decentralized coordination and reduced network traffic on a single node can perform better than the centralized coordinator Raft consensus protocol. The performance was similar in both Canopus and Raft with few inconsistencies in Canopus after 8 orderer nodes in transfer-money smart contract. The possible reasons will be discussed in chapter 6.

Kafka is the poorest performer here. Raft will mostly replace Kafka in the nearby future due to its increasing popularity, simplicity and stable performance. Many permissioned blockchains have already adapted Raft or in the process of developing support for it.

Chapter 6 - Conclusion

6.1 Introduction

The aim of this research was to provide more scalability performance of blockchains by introducing a decentralized consensus algorithm in order to overcome the bottlenecks of current consensus algorithms adapted in blockchains. Through this research a permissioned distributed ledger was designed with a decentralized consensus algorithm called Canopus and implemented using the Hyperledger Fabric blockchain framework and evaluated for performance in terms of transaction throughput and latency.

We considered all aspects of blockchain and how each process in blockchain effects scalability. Among these processes we chose to optimize the consensus algorithm which is responsible for making total order of transactions in a network of nodes. Then we critically analyzed both permissionless blockchains and permissioned blockchains and the consensus algorithms they've adapted. Considering the time frame of the research, developing and testing a consensus mechanism in a permissionless blockchain was not practical. Therefore, we settled on optimizing the consensus in permissioned blockchains. Given CFT priority, we analyzed the consensus algorithms adapted in permissioned blockchains, looked into the bottlenecks in these methods and proposed a decentralized consensus protocol called Canopus that can overcome them.

Subsequently the popular consensus algorithms and the proposed Canopus consensus were evaluated in the Hyperledger Fabric blockchain framework in terms of transaction throughput and latency.

6.2 Conclusions About Research Questions

We have analyzed the blockchain platform and identified the processes that limit scalability of the blockchain. Among them was the consensus mechanism in blockchains. After the analysis of consensus mechanism in permissioned blockchains, we attempted to optimize the performance by introducing a decentralized consensus algorithm called Canopus. The algorithm was tested by using a framework that supported pluggable consensus such as Hyperledger Fabric. However, the results were not as expected. The Raft consensus algorithm performed slightly better than Canopus. But there can be many reasons for this. The test was carried out in a single virtual machine using Amazon EC2. To get the advantages of network bottleneck over Raft, the test should have been conducted in different data centers. This is how Canopus was supposed to be implemented. Due to unavailability of the resources, this was not tested. The other reason for less performance is the developed Canopus algorithm may not have been well optimized. Due to the limited time frame, all variables to optimize the algorithm could not have been covered. Raft on the other hand is open source and developed by many professionals in the field continuously. It is at the peak performance as it is intended in production environments. The workforce difference on these 2 implementations is massive and therefore the comparison is somewhat at a disadvantage in this research. The conclusion that is arrived after observing the evaluation is that a decentralized consensus mechanism cannot scale blockchains. However, if more time is provided to refine the algorithm, it can be possible to perform better than existing consensus protocols.

The next section on limitation and further work would present future prospects and possibilities for optimizing the consensus mechanism and maybe adapt Canopus to be Byzantine fault tolerant.

6.3 Limitations and Implications for Further Research

There were several features in Canopus that was not implemented for this research due to the limited time frame. Also many more optimizations could have been made that might have been overlooked during the design and implementation of the consensus protocol. These features and optimizations maybe carried out as future prospects for increasing scalability in blockchains.

Since we adapted Hyperledger Fabric as the blockchain framework to test out Canopus, the implementation was done over the execute-order-validate architecture. There can be room for improvement on different architectures in blockchains that support pluggable consensus.

The blockchain network was deployed as a network of Docker containers on a virtual machine instance. This implies the CPU power is divided among all the peer and orderer nodes. This can be speculated as another reason for lower throughput in all consensus algorithms. Provided enough CPU power, there is the possibility of Canopus consensus to overcome the performance of Raft.

Canopus can be adapted to support Byzantine fault tolerance. A theoretical paper proposed this algorithm as RCanopus [45] and it can be implemented over the Canopus protocol which can benefit to development and adaptation of blockchains in untrusting environments.

References

- [1] Distributed Ledgers, article. <https://www.investopedia.com/terms/d/distributed-ledgers.asp>
- [2] C. Cachin and M. Vukolić. “Blockchain consensus protocols in the wild,” A. W. Richa, editor, 31st Intl. Symposium on Distributed Computing (DISC 2017), pages 1:1–1:16, 2017.
- [3] Bartoletti, M. and Pompianu, L., 2017, April. An empirical analysis of smart contracts: platforms, applications, and design patterns. In International conference on financial cryptography and data security (pp. 494-509). Springer, Cham.
- [4] Cachin, C., 2016, July. Architecture of the hyperledger blockchain fabric. In Workshop on distributed cryptocurrencies and consensus ledgers (Vol. 310, p. 4).
- [5] Baliga, A., 2017. Understanding blockchain consensus models. Persistent, 2017(4), pp.1-14.
- [6] Nakamoto, S., 2019. Bitcoin: A peer-to-peer electronic cash system. Manubot.
- [7] Im, D.K.D., 2018. The Blockchain Trilemma.
- [8] Wood, G., 2014. Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper, 151(2014), pp.1-32.
- [9] Making sense of blockchain smart contracts, <https://www.coindesk.com/making-sense-smart-contracts>.
- [10] XRP, <https://ripple.com/xrp/>
- [11] Tether, <https://tether.to/>
- [12] Smart contracts: The good, the bad and the lazy, <http://www.multichain.com/blog/2015/11/smart-contracts-good-bad-lazy/>. Last accessed 2017/01/14
- [13] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y. and Muralidharan, S., 2018,

April. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Proceedings of the Thirteenth EuroSys Conference (pp. 1-15).

[14] E. Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. M.Sc. Thesis, University of Guelph, Canada, June 2016.

[15] M. Hearn. Corda: A distributed ledger. Available online, https://docs.corda.net/_static/corda-technical-whitepaper.pdf, 2016.

[16] W.Martino. Kadena—the first scalable, high performance private blockchain.Whitepaper, <http://kadena.io/docs/Kadena-ConsensusWhitePaper-Aug2016.pdf>, 2016.

[17] M. Vukolić. Quorum Systems: With Applications to Storage and Consensus. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2012.

[18] Chauhan, A., Malviya, O.P., Verma, M. and Mor, T.S., 2018, July. Blockchain and scalability. In 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C) (pp. 122-128). IEEE.

[19] Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Sirer, E.G. and Song, D., 2016, February. On scaling decentralized blockchains. In International conference on financial cryptography and data security (pp. 106-125). Springer, Berlin, Heidelberg.

[20] Bano, S., Al-Bassam, M. and Danezis, G., 2017. The road to scalable blockchain designs. USENIX; login: magazine.

[21] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, “Bitcoin-NG: A Scalable Blockchain Protocol,” in Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (NSDI ’16), pp. 45–59: <http://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-eyal.pdf>.

[22] Gomez, Miguel. “Ethereum Co-Founder Vitalik Buterin Weighs in on Blockchain Improvement & Scaling Issues.” Cryptovest , Cryptovest, 27 Nov. 2017, cryptovest.com/news/ethereum-co-founder-vitalik-buterin-weighs-in-on-blockchain-improvement--scaling-issues/.

- [23] F. B. Schneider. Implementing fault-tolerant services using the state-machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.
- [24] Lamport, L., 2001. Paxos made simple. *ACM Sigact News*, 32(4), pp.18-25.
- [25] B. M. Oki and B. Liskov. Viewstamped replication: A new primary copy method to support highly available distributed systems. In *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, 1988.
- [26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internetscale systems. In *Proc. USENIX Annual Technical Conference*, 2010.
- [27] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proc. 41st International Conference on Dependable Systems and Networks*, 2011.
- [28] In Search of an Understandable Consensus Algorithm Diego Ongaro and John Ousterhout Stanford University
- [29] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.
- [30] Sousa, J., Bessani, A. and Vukolic, M., 2018, June. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)* (pp. 51-58). IEEE.
- [31] Ethers, source code. <https://github.com/ethereum/wiki/wiki/Ethers>
- [32] Litecoin, open source P2P digital currency. <https://litecoin.org>.
- [33] Sukhwani, H., Martínez, J.M., Chang, X., Trivedi, K.S. and Rindos, A., 2017, September. Performance modeling of pbft consensus process for permissioned blockchain network (hyperledger fabric). In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)* (pp. 253-255). IEEE.
- [34] Rizvi, S., Wong, B. and Keshav, S., 2017, November. Canopus: A scalable and massively parallel consensus protocol. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies* (pp. 426-438).

- [35] The Hyperledger Whitepaper Working Group. An introduction to Hyperledger. Volume 1.1. 2018.
- [36] Hyperledger Architecture Working Group. Hyperledger Architecture, Volume 1 Introduction to Hyperledger Business Blockchain Design Philosophy and Consensus. Volume 1. 2017.
- [37] Hyperledger Fabric, source code. <https://github.com/hyperledger/fabric>
- [38] Blockchain technology for healthcare,
<https://www.changehealthcare.com/innovation/blockchain?fbclid=IwAR06HHF4c5z--UuCyEvAxxpPzjFsWlhQsADp9WyDLZYcGpBbYEhQ2L8WZQ0>
- [39] Everledger, https://www.everledger.io/?fbclid=IwAR3KS1UPv9A4IOgduKyRaEfN0V3ArIQMPn-Rh4_JEc5IOVwIT0N_iONW3Gs
- [40] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. J. Mullender, editor, Distributed Systems (2nd Ed.). ACM Press & Addison-Wesley, New York, 1993. Expanded version appears as Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994 (atomic broadcast)
- [41] QuorumChain consensus, source code.
<https://github.com/jpmorganchase/quorum/wiki/QuorumChain-Consensus-%5BQuorum-v1.x-ONLY%5D>
- [42] King, S. and Nadal, S., 2012. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. self-published paper, August, 19.
- [43] Bessani, A., Sousa, J. and Alchieri, E.E., 2014, June. State machine replication for the masses with BFT-SMaRt. In 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (pp. 355-362). IEEE.
- [44] André Allavena, Qiang Wang, Ihab Ilyas, and Srinivasan Keshav. 2006. LOT: A robust overlay for distributed range query processing. Technical Report. CS-2006- 21, University of Waterloo.

- [45] Keshav, S., Golab, W., Wong, B., Rizvi, S. and Gorbunov, S., 2018. RCanopus: Making canopus resilient to failures and byzantine faults. *arXiv preprint arXiv:1810.09300*. Keshav, S., Golab, W., Wong, B., Rizvi, S. and Gorbunov, S., 2018. RCanopus: Making canopus resilient to failures and byzantine faults. arXiv preprint arXiv:1810.09300.
- [46] Hyperledger Caliper, source code. <https://github.com/hyperledger/caliper>
- [47] Nasir, Q., Qasse, I.A., Abu Talib, M. and Nassif, A.B., 2018. Performance analysis of hyperledger fabric platforms. *Security and Communication Networks*, 2018.

Appendix C

Appendix C.1

Configtx.yaml

```
Organizations:
  - &OrdererOrg
    Name: OrdererOrg
    ID: OrdererMSP
    MSPDir: crypto-config/ordererOrganizations/example.com/msp
    Policies:
      Readers:
        Type: Signature
        Rule: "OR('OrdererMSP.member')"
      Writers:
        Type: Signature
        Rule: "OR('OrdererMSP.member')"
      Admins:
        Type: Signature
        Rule: "OR('OrdererMSP.admin')"
  - &Org1
    Name: Org1MSP
    ID: Org1MSP
    MSPDir: crypto-config/peerOrganizations/org1.example.com/msp
    Policies:
      Readers:
        Type: Signature
        Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.client')"
      Writers:
        Type: Signature
        Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
      Admins:
        Type: Signature
        Rule: "OR('Org1MSP.admin')"
```

Appendix C.2

Crypto-config.yaml

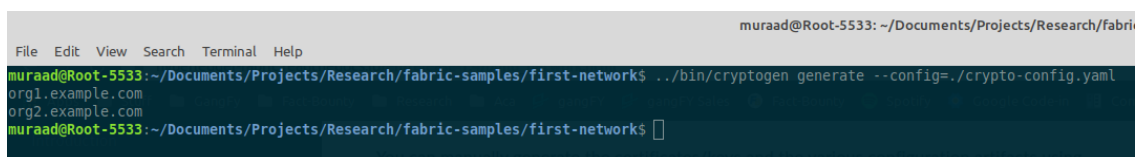
```
OrdererOrgs:
  - Name: Orderer
    Domain: example.com
    EnableNodeOUs: true

    Specs:
      - Hostname: orderer
      - Hostname: orderer2
      - Hostname: orderer3
      - Hostname: orderer4
      - Hostname: orderer5

PeerOrgs:
  - Name: Org1
    Domain: org1.example.com
    EnableNodeOUs: true

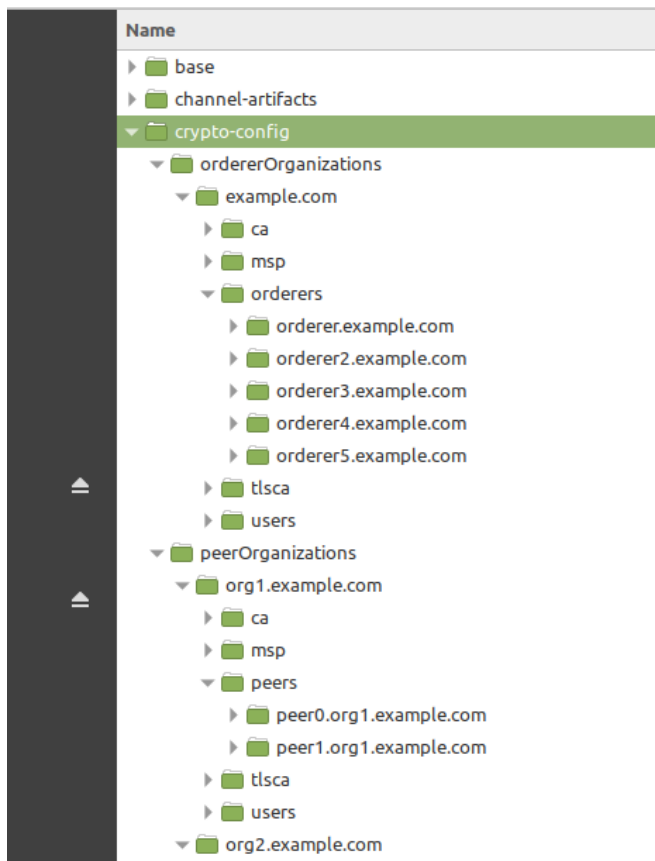
    Template:
      Count: 2
    Users:
      Count: 1
  # -----
  # Org2: See "Org1" for full specification
  # -----
  - Name: Org2
    Domain: org2.example.com
    EnableNodeOUs: true
    Template:
      Count: 2
    Users:
      Count: 1
```

Command to build cryptographic certificates



```
muraad@Root-5533: ~/Documents/Projects/Research/fabric
File Edit View Search Terminal Help
muraad@Root-5533:~/Documents/Projects/Research/fabric-samples/first-network$ ../bin/cryptogen generate --config=./crypto-config.yaml
org1.example.com
org2.example.com
muraad@Root-5533:~/Documents/Projects/Research/fabric-samples/first-network$
```

Directories created to separate certificates for each organization and node



Appendix C.3

Orderer related parameters

```
Orderer: &OrdererDefaults

OrdererType: solo

Addresses:
- orderer.example.com:7050

BatchTimeout: 2s

BatchSize:
  MaxMessageCount: 10
  AbsoluteMaxBytes: 99 MB
  PreferredMaxBytes: 512 KB

Kafka:
  Brokers:
  - 127.0.0.1:9092

Canopus:

Consenters:
- Host: orderer.example.com
  Port: 7050
  ClientTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/server.crt
  ServerTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/server.crt
- Host: orderer2.example.com
  Port: 7050
  ClientTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer2.example.com/tls/server.crt
  ServerTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer2.example.com/tls/server.crt
- Host: orderer3.example.com
  Port: 7050
  ClientTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer3.example.com/tls/server.crt
  ServerTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer3.example.com/tls/server.crt
- Host: orderer4.example.com
  Port: 7050
  ClientTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer4.example.com/tls/server.crt
  ServerTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer4.example.com/tls/server.crt
- Host: orderer5.example.com
  Port: 7050
  ClientTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer5.example.com/tls/server.crt
  ServerTLS Cert: crypto-config/ordererOrganizations/example.com/orderers/orderer5.example.com/tls/server.crt

# Organizations is the list of orgs which are defined as participants on
# the orderer side of the network
```

Canopus and channel profiles

```
#####
Profiles:

TwoOrgsChannel:
  Consortium: SampleConsortium
  <<: *ChannelDefaults
  Application:
    <<: *ApplicationDefaults
    Organizations:
      - *Org1
      - *Org2
    Capabilities:
      <<: *ApplicationCapabilities

Canopus:
  <<: *ChannelDefaults
  Capabilities:
    <<: *ChannelCapabilities
  Orderer:
    <<: *OrdererDefaults
    OrdererType: canopus
Canopus: You, a few seconds ago • Uncommitted changes
Consenters:
- Host: orderer.example.com
  Port: 7050
  ClientTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/server.crt
  ServerTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/server.crt
- Host: orderer2.example.com
  Port: 7050
  ClientTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer2.example.com/tls/server.crt
  ServerTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer2.example.com/tls/server.crt
- Host: orderer3.example.com
  Port: 7050
  ClientTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer3.example.com/tls/server.crt
  ServerTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer3.example.com/tls/server.crt
- Host: orderer4.example.com
  Port: 7050
  ClientTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer4.example.com/tls/server.crt
  ServerTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer4.example.com/tls/server.crt
- Host: orderer5.example.com
  Port: 7050
  ClientTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer5.example.com/tls/server.crt
  ServerTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer5.example.com/tls/server.crt
Addresses:
- orderer.example.com:7050
- orderer2.example.com:7050
- orderer3.example.com:7050
- orderer4.example.com:7050
- orderer5.example.com:7050
```

Appendix C.4

Terminal command and output to create fabric packages and orderer Docker image

```
~/Projects/Research/src/github.com/hyperledger/fabric | canopus-v0.1 | make clean configtxgen configtxlator cryptogen idemixgen peer orderer orderer-docker
docker images --quiet --filter=reference=hyperledger/fabric-peer:amd64-1.4.5-snapshot-* | xargs docker rmi -f
docker images --quiet --filter=reference=hyperledger/fabric-orderer:amd64-1.4.5-snapshot-* | xargs docker rmi -f
Untagged: hyperledger/fabric-orderer:amd64-1.4.5-snapshot-a66c78119
Untagged: hyperledger/fabric-orderer:amd64-latest
Untagged: hyperledger/fabric-orderer:latest
Deleted: sha256:84bcd304efd666bc553f4971147db0703a75d6f6e0254b836216a6481aea746f
Deleted: sha256:0c60f460378b4364c05b783641bb78679782232b238d6d65051327751cc12933
Deleted: sha256:806d3b5466139233d8fe0c7fa701b5b4f6e09e30433eb05ee240fa5a156d7165
Deleted: sha256:8fc68223affce662fb19156f01a05f72676b9e30ff4dbc00169812e5db80a7b9
Deleted: sha256:51a94dac68a5166567d9f15e07e28f2e1d6298dc498095e0c86d0059ff765db0
Deleted: sha256:5b6d60f1fdeb640b12c1971991f2ce5827cac97a2f681123b2667ba1af78dd9
Deleted: sha256:f4556a7a61110f0781c8e9eb0a832543d68c5d758f414f4e1d3a9d04f05f065
Deleted: sha256:4a9fd8628fea44719271e54f3af99b135135f0d04ad76e59b42346353a6821f2
Deleted: sha256:abb0583905ce45248c515ebd6119b994b32331bbfb55f64581fe71e708801e96
Deleted: sha256:8087816a40ddd0b3fa7dff50df109c37a3321dbf6cc58b85ab323004538ac989
docker images --quiet --filter=reference=hyperledger/fabric-ccenv:amd64-1.4.5-snapshot-* | xargs docker rmi -f
docker images --quiet --filter=reference=hyperledger/fabric-buildenv:amd64-1.4.5-snapshot-* | xargs docker rmi -f
docker images --quiet --filter=reference=hyperledger/fabric-tools:amd64-1.4.5-snapshot-* | xargs docker rmi -f
cd unit-test && docker-compose down
WARNING: The TEST_PKG variable is not set. Defaulting to a blank string.
WARNING: The JOB_TYPE variable is not set. Defaulting to a blank string.
.build/bin/configtxgen
CGO_FLAGS="" GOBIN=/Users/root5533/Projects/Research/src/github.com/hyperledger/fabric/.build/bin go install -tags "" -ldflags "-X github.com/hyperledger/fabric/common/tool
Binary available as .build/bin/configtxgen
.build/bin/configtxlator
CGO_FLAGS="" GOBIN=/Users/root5533/Projects/Research/src/github.com/hyperledger/fabric/.build/bin go install -tags "" -ldflags "-X github.com/hyperledger/fabric/common/tool
ator
Binary available as .build/bin/configtxlator
.build/bin/cryptogen
CGO_FLAGS="" GOBIN=/Users/root5533/Projects/Research/src/github.com/hyperledger/fabric/.build/bin go install -tags "" -ldflags "-X github.com/hyperledger/fabric/common/tool
Binary available as .build/bin/cryptogen
.build/bin/idemixgen
CGO_FLAGS="" GOBIN=/Users/root5533/Projects/Research/src/github.com/hyperledger/fabric/.build/bin go install -tags "" -ldflags "-X github.com/hyperledger/fabric/common/tool
Binary available as .build/bin/idemixgen
.build/bin/peer
CGO_FLAGS="" GOBIN=/Users/root5533/Projects/Research/src/github.com/hyperledger/fabric/.build/bin go install -tags "" -ldflags "-X github.com/hyperledger/fabric/common/meta
com/hyperledger/fabric/common/metadata.BaseVersion=0.4.18 -X github.com/hyperledger/fabric/common/metadata.BaseDockerLabel=org.hyperledger.fabric -X github.com/hyperledger/fa
BaseDockerNamespace=hyperledger" github.com/hyperledger/fabric/peer
Binary available as .build/bin/peer
.build/bin/orderer
CGO_FLAGS="" GOBIN=/Users/root5533/Projects/Research/src/github.com/hyperledger/fabric/.build/bin go install -tags "" -ldflags "-X github.com/hyperledger/fabric/common/meta
com/hyperledger/fabric/common/metadata.BaseVersion=0.4.18 -X github.com/hyperledger/fabric/common/metadata.BaseDockerLabel=org.hyperledger.fabric -X github.com/hyperledger/fa
BaseDockerNamespace=hyperledger" github.com/hyperledger/fabric/orderer
Binary available as .build/bin/orderer
Building .build/docker/bin/orderer
```

Appendix C.5

When receiving message from orderer node or client node

```
934 // for canopus
935 case n := <-c.receivedFrom: You, a month ago * canopus pass simple loadtest.sh
936     if n == uint64(c.currentNode) {
937         batches, pending, err := c.CanpCreateBatch(c.currentNode)
938         if err != nil {
939             c.logger.Errorf("Failed to order message: %s", err)
940             continue
941         }
942         if pending {
943             startCanpTimer()
944         } else {
945             if len(batches) != 0 {
946                 stopCanpTimer()
947                 c.currentNode = nextNode(c.currentNode)
948                 c.logger.Infof("Came to create full batch, next node : ", c.currentNode)
949                 for _, batch := range batches {
950                     block := c.support.CreateNextBlock(batch)
951                     c.support.WriteBlock(block, nil)
952                     c.logger.Infof("Writing block [%d] to ledger without cut", block.Header.Number)
953                     c.Metrics.CommittedBlockNumber.Set(float64(block.Header.Number))
954                 }
955                 go c.scanNextBuffer()
956             }
957         }
958     } else {
959         c.currentNode = int(n)
960     }
961
```

When block interval time has expired

```
962 case <-canpTimer.C():
963     canpTicking = false
964     batch := c.support.BlockCutter().Cut()
965     block := c.support.CreateNextBlock(batch)
966     c.support.WriteBlock(block, nil)
967     c.logger.Infof("Writing block [%d] to ledger with cut", block.Header.Number)
968     c.Metrics.CommittedBlockNumber.Set(float64(block.Header.Number)) You, a month ago *
969 }
970 }
971 }
972
```

Finalize block execution

```
978 func (c *Chain) writeBlock(block *common.Block, index uint64) {
979     if block.Header.Number > c.lastBlock.Header.Number+1 {
980         c.logger.Panicf("Got block [%d], expect block [%d]", block.Header.Number, c.lastBlock.Header.Number+1)
981     } else if block.Header.Number < c.lastBlock.Header.Number+1 {
982         c.logger.Infof("Got block [%d], expect block [%d], this node was forced to catch up", block.Header.Number, c.lastBlock.Header.Number)
983         return
984     }
985
986     if c.blockInflight > 0 {
987         c.blockInflight-- // only reduce on leader
988     }
989     c.lastBlock = block
990
991     c.logger.Infof("Writing block [%d] (Raft index: %d) to ledger", block.Header.Number, index)
992
993     if utils.IsConfigBlock(block) {
994         c.writeConfigBlock(block, index)
995         return
996     }
997
998     c.raftMetadataLock.Lock()
999     c.opts.BlockMetadata.RaftIndex = index
1000     m := utils.MarshalOrPanic(c.opts.BlockMetadata)
1001     c.raftMetadataLock.Unlock()
1002
1003     c.support.WriteBlock(block, m)
1004 }
```

Each message received is ordered and validated

```
1012 func (c *Chain) ordered(msg *orderer.SubmitRequest) (batches []*common.Envelope, pending bool, err error) {
1013     seq := c.support.Sequence()
1014
1015     if c.isConfig(msg.Payload) {
1016         // ConfigMsg
1017         if msg.LastValidationSeq < seq {
1018             c.logger.Warnf("Config message was validated against %d, although current config seq has advanced (%d)", msg.LastValidationSeq, seq)
1019             msg.Payload, _, err = c.support.ProcessConfigMsg(msg.Payload)
1020             if err != nil {
1021                 c.Metrics.ProposalFailures.Add(1)
1022                 return nil, true, errors.Errorf("bad config message: %s", err)
1023             }
1024
1025             if err = c.checkConfigUpdateValidity(msg.Payload); err != nil {
1026                 c.Metrics.ProposalFailures.Add(1)
1027                 return nil, true, errors.Errorf("bad config message: %s", err)
1028             }
1029         }
1030         batch := c.support.BlockCutter().Cut()
1031         batches = []*common.Envelope{batch}
1032         if len(batches) != 0 {
1033             batches = append(batches, batch)
1034         }
1035         batches = append(batches, []*common.Envelope(msg.Payload))
1036         return batches, false, nil
1037     }
1038     // it is a normal message
1039     if msg.LastValidationSeq < seq {
1040         c.logger.Warnf("Normal message was validated against %d, although current config seq has advanced (%d)", msg.LastValidationSeq, seq)
1041         if _, err := c.support.ProcessNormalMsg(msg.Payload); err != nil {
1042             c.Metrics.ProposalFailures.Add(1)
1043             return nil, true, errors.Errorf("bad normal message: %s", err)
1044         }
1045     }
1046     batches, pending = c.support.BlockCutter().Ordered(msg.Payload)
1047     return batches, pending, nil
1048 }
1049 }
```