BYZANTINE FAULT TOLERANT CONSENSUS FOR HYPERLEDGER FABRIC

by

Ahmed Al Salih

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2024

Approved by:

Dr. Yongge Wang

Dr. Chao Wang

Dr. Jinpeng Wei

Dr. Xi (Sunshine) Niu

Dr. M. Yasin Akhtar Raja

©2024
Ahmed Al Salih
ALL RIGHTS RESERVED

ABSTRACT

AHMED AL SALIH. Byzantine Fault Tolerant Consensus For Hyperledger Fabric.
(Under the direction of DR. YONGGE WANG)

Decentralized systems play a crucial role in both the public and private sectors, addressing a wide range of organizational needs. At the core of these systems is the dissemination protocol. Hyperledger Fabric is a leading platform for production-ready distributed network systems. Although Hyperledger Fabric is designed to support pluggable consensus protocols, it needs more detailed technical guidance on integrating new consensus modules. Initially, Fabric employed Kafka as its consensus protocol but later transitioned to Raft. Both Kafka and Raft are Crash Fault-Tolerant (CFT) protocols that do not account for Byzantine fault-tolerant participants. This research explores the necessary steps to integrate a consensus protocol into Hyperledger Fabric, focusing specifically on incorporating the Byzantine Fault Tolerant (BFT) BDLS protocol. Our proposed BFT solution, inspired by the initial Dwork, Lynch, and Stockmeyer (DLS) protocol and adapted as the Blockchain DLS (BDLS) protocol, is recognized as one of the most efficient and promising BFT protocols for blockchain systems. This study provides a comprehensive technical analysis of integrating BDLS into Hyperledger Fabric, highlighting the complexities and advantages of this integration. Chapter five presents a performance comparison between Raft-based and BDLS-based Hyperledger Fabric. The findings demonstrate that Hyperledger Fabric, when utilizing the BDLS protocol, achieves performance levels comparable to those of the Raft-based Fabric. In 2024, Hyperledger Fabric announced a new BFT solution, SmartBFT, in the beta release of Fabric version 3.0. However, SmartBFT, which originates from the Practical Byzantine Fault Tolerance (PBFT) protocol, faces significant scalability challenges due to its high message complexity. This complexity severely impacts network efficiency, particularly in large-scale deployments in sectors

such as healthcare, finance, and education. Our research shows that the message complexity of SmartBFT increases quadratically with the number of ordering nodes, resulting in substantial communication overhead. In contrast, our proposed BDLS protocol maintains linear message complexity, making it more scalable and efficient for large networks. Furthermore, our research proposes a solution to enhance the security of IoT-Edge servers and Cloud replicas within the new Fabric-BDLS framework. The experimental results indicate that BDLS provides consistent performance advantages. Throughput analysis of Fabric 3.0 reveals a significant performance drop for SmartBFT, achieving only 40% and 20% of Raft's throughput in LAN and WAN environments, respectively. Conversely, BDLS-based Fabric achieves 90% to 95% of Raft's throughput, underscoring its superior scalability and efficiency and instilling confidence in its suitability for large-scale deployments. All the codes are available in the GitHub repository: `https://github.com/BDLS-bft/fabric`.

ACKNOWLEDGEMENTS

Portions of this thesis are reprinted, with permission, from Ahmed Al Salih and Yongge Wang, Securing the Connected World: Fast and Byzantine Fault Tolerant Protocols for IoT, Edge, and Cloud Systems, 2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW), May 2024. ©2024 IEEE.

DOI: https://doi.org/10.1109/CCGridW63211.2024.00010

Portions of this thesis are reprinted, with permission, from Ahmed Al Salih and Yongge Wang, BDLS as a Blockchain Finality Gadget: Improving Byzantine Fault Tolerance in Hyperledger Fabric, IEEE Access, ©2024 IEEE.

DOI: https://doi.org/10.1109/ACCESS.2024.3481319

Portions of this thesis includes material reprinted with permission from Ahmed Al Salih and Yongge Wang, Pluggable Consensus in Hyperledger Fabric, BIOTC 2024, ©2024 ACM.

ACM ISBN 979-8-4007-1700-0/24/07.

DOI: https://doi.org/10.1145/3688225.3688237

Portions of this thesis are reprinted, with permission, from Ahmed Al Salih and Yongge Wang, Innovative Byzantine Fault Tolerance in Hyperledger Fabric with BDLS ,2024 Sixth International Conference on Blockchain Computing and Applications (BCCA), November 2024. ©2024 IEEE.

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of The University of North Carolina at Charlotte's products or services. Internal or personal use of this material is permitted.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODE LISTINGS

LIST OF ABBREVIATIONS

BFT An acronym for Byzantine fault tolerance.

CFT An acronym for Crash fault tolerance.

Chaincode An acronym for smart contracts in Hyperledger Fabric system.

DLT An acronym for Distributed Ledger Technology.

gRPC An acronym for Google Remote Procedure Call.

HLF An acronym for Hyperledger Fabric.

MSP An acronym for Membership Services Provider.

Orderer An acronym for the Ordering service in Hyperledger Fabric.

OSN An acronym for Ordering Service node.

PBFT An acronym for Practical Byzantine Fault Tolerance.

PoW An acronym for Proof-of-Work.

Quorum An acronym for the minimum number of Orderer nodes necessary to be active to achieve consensus.

TPS Transactions Per Second. The number of transactions a blockchain network can process within a second.

WAL An acronym for Write-ahead logging.

CHAPTER 1: INTRODUCTION

The enterprise blockchain market, which includes permissioned platforms like Hyperledger Fabric, is projected to grow significantly from **$4.9 billion in 2021** to an estimated **$50 billion by 2030**, with a compound annual growth rate (CAGR) of **33.6%** reported by Fortune Business Insights [1] and MarketsandMarkets [2].This rapid expansion highlights the increasing adoption of blockchain technology in various industries, driven by its ability to provide secure, decentralized, and efficient solutions for business operations. As enterprises scale their blockchain deployments, the demand for robust consensus mechanisms, such as Byzantine Fault Tolerant (BFT) models, will continue to grow to ensure system security, trust, and fault tolerance.

In specific sectors, blockchain adoption is also accelerating. For instance, the blockchain market within the supply chain sector is expected to reach **$15 billion by 2028**, as organizations increasingly rely on distributed ledger technologies to enhance transparency, security, and traceability across global supply chains (Allied Market Research, 2023) [3].

Similarly, the healthcare sector is forecasted to see blockchain integration grow to **$5.61 billion by 2027**, driven by the need for secure and tamper-resistant systems for managing patient data and pharmaceutical supply chains (ResearchAndMarkets, 2023) [4].

These projections underscore the critical role of secure consensus protocols in ensuring the reliability and scalability of permissioned blockchain networks like Hyperledger Fabric in enterprise applications.

Hyperledger Fabric Orderer system plays a crucial role in ensuring the consistency and reliability of transactions within a Hyperledger Fabric network. The selection

of an appropriate consensus protocol is paramount to achieving these objectives. This research delves into the technical aspects of implementing the mechanism for ordering Hyperledger Fabric with an emphasis on the Raft consensus. Furthermore, it explores the steps required to plug any consensus protocol into the Hyperledger Fabric Orderer service, exemplifying this process by integrating the Byzantine Fault Tolerance protocol BDLS [5]. The Raft protocol [6], known for its simplicity and crash fault-tolerance characteristics, has gained prominence in distributed systems. By understanding its inner workings, participants can grasp how it enhances the resiliency and performance of the Orderer system. Additionally, exploring the steps required to integrate the BDLS protocol is a foundation for seamlessly incorporating other consensus protocols. This research will go beyond theoretical concepts and provide practical insights into technical implementation. We will share our firsthand experience integrating the BDLS consensus protocol, proposing precise steps for the integration phases. By showcasing this integration, the reader will better understand the intricacies and advantages of combining alternative consensus protocols with the Hyperledger Fabric Orderer system. Overall, this research provides a comprehensive knowledge of the technical implementation of the Hyperledger Fabric Orderer system, with a particular focus on the Raft protocol. By demonstrating the integration of the BDLS protocol, readers will gain valuable insights into the steps required to plug any consensus protocol into the Hyperledger Fabric Orderer service.

Furthermore,In this research we presents a comprehensive analysis of the BDLS consensus protocol, with a focus on its application in blockchain systems. The key contributions include the introduction of an optimized message complexity model that enhances the scalability of BDLS in comparison to traditional Byzantine Fault Tolerant (BFT) protocols. Additionally, we provide detailed performance results, demonstrating that BDLS achieves a significantly higher throughput in terms of transactions per second (TPS) under realistic network conditions. Specifically, our experiments

demonstrate that BDLS reaches 90% to 95% of Raft's TPS, outperforming existing BFT solutions in similar environments. These results underline the potential of BDLS for improving the efficiency of large-scale, production-grade blockchain deployments.

At the end of the introduction, this thesis is structured as follows: Chapter 1 provides a comprehensive overview of blockchain technology, with a focus on Hyperledger Fabric, including its architecture, components, transaction flow, and consensus protocols. It also explores security issues and relevant attacks, concluding with a review of related work. Chapter 2 delves into the structure and function of the Fabric Orderer, detailing the history of consensus algorithms and the mechanics of the Raft consensus. Chapter 3 presents the core contribution of this research, which involves integrating the Byzantine Fault Tolerant BDLS protocol into Hyperledger Fabric by examining BDLS's integration and implementation within Fabric, including code changes and network setup. Chapter 4 focuses on securing IoT, Edge, and Cloud systems through the use of BDLS, addressing performance and security challenges in these environments. Chapter 5 evaluates the performance of BDLS within Hyperledger Fabric through a series of experiments comparing it to Raft and SmartBFT, highlighting BDLS's superior scalability and efficiency in large-scale deployments. Chapter 6 presents the complete BDLS-Fabric architecture. Finally, Chapter 7 presents the conclusions drawn from this research, summarizing the key findings and discussing their implications for future work in the field.

1.1 Literature Review

The proliferation of technology has enabled several trends, including blockchain technology, which is very promising for developing decentralized networks where participants can cooperate to verify and endorse each transaction instead of relying on central authority. Blockchain technology brings profound benefits to businesses and individuals that could improve human lifestyles in many industrial sectors, such as finance transactions, health, and supply chains. A recent study indicated that the Hyperledger Fabric framework is one of the frameworks that has witnessed a noticeable growth among open source developers [7]. The decentralized technology relies on distributed ledgers that are managed by a group of nodes (peers) in lieu of a centralized authority that validates new transactions in the chaincode. By the end of 2024, it is expected that corporations will spend approximately $20 billion US dollars per year towards blockchain technical services [7]. The decentralized mechanism eliminates the necessity of a trusted intermediary entity [8] at the same time, it is widely distributed among the networks, ensuring the immutability, traceability, and trustworthiness of every newly added transaction. Based on Forbes Blockchain 50 2021 in their third annual, Blockchain 50 recognizes companies that have at least $1 billion in revenue or market capitalization who are pioneers in the use of distributed ledger blockchain [9]. In fact, the Hyperledger Fabric is permissioned to enable nodes to confirm which one approves the newly added transactions. It is still an open source to enable researchers and practitioners to design their own customized smart contracts. This brings several benefits: nodes run tamper-proof consensus algorithms, broadcast transactions to other network peers, and update the chaincode state. Despite the great benefits of the decentralized. This chapter objective is to shed light on the current security challenges in decentralized blockchain technology, specifically focusing on investigating the security attacks and threats impacting the Hyperledger Fabric framework. The purpose of this chapter is to discuss and investigate current litera-

ture on permissioned blockchain, specifically Hyperledger Fabric security challenges. We further review the security approaches comparatively, discuss their advantages, and highlight their shortcomings. In addition, due to the variety of security research problems related to the consensus protocols, we aim to present a comparative study of the consensus protocols. We also focused on the partially synchronized network model such as Raft, which serves as a Hyperledger Fabric consensus protocol. Our study provides insight on several relevant security threats, including Denial of Service attacks (DoS), and discusses a literature review on a possible solution.

The rest of this chapter as follows: section 2 introduces the Hyperledger Fabric major concepts. In section 3, we shed some light on Hyperledger Fabric vulnerabilities and the impact on the HLF system. In section 4, we further discuss the attacks that impact Hyperledger itself. Then, in section 5, continued efforts are needed to investigate new directions and open problems that are worth examining. Finally, in section 6, we conclude the literature review chapter.

1.2 Hyperledger

Hyperledger is an open-source project that maintains a distributed ledger [10] typically implemented within a peer-to-peer network [8] and code base [11]. Hyperledger developed several frameworks as a business blockchain. All the frameworks are open source and for general uses, implying that the business is free to store any form of asset that serves the organization's needs, except for Hyperledger Indy, which is a framework for decentralized identity management. Hyperledger is a collaborative effort that maintains a ledger of record as it is named "blocks". The blocks contain the transactions per that particular asset [12]. The blocks are safeguarded against tampering. The cryptographic hashes, as well as a consensus process, prevent blocks against manipulation. To provide an adequate definition of the Blockchain, which is an immutable ledger that records transactions in a distributed network [13]. In the different use cases, distributed ledgers may have radically different needs. When participants

have a high level of confidence, such as between financial institutions with Human Resources, blockchains can use a faster consensus process to add blocks to the chain with quicker confirmation periods. When there is less trust between players, on the other hand, they must accept slower processing for additional security. Hyperledger supports a wide range of use cases [14]. The members of Hyperledger's networks are called peers [13]. Peers normally in a Hyperledger network are untrusting peers. Each individual peer maintains a copy of the ledger. In order to validate the transactions, the peer uses a consensus protocol for validation [13]. The type of participation that allows any peers within the network to participate is called permissionless or public. Based on the economic incentives or the (PoW) proof of work, public networks using native cryptocurrency also involve a consensus. The Cryptocurrency trading platform are open and decentralized. Therefore, the ownership of the actual Cryptocurrency transfers from one to another is a permissionless blockchain [15] as Bitcoin [8] and Ethereum [16]

It is now well established that the nature of Blockchain security remains unclear. However, the influence of Hyperledger Fabric on Blockchain has brought potential solutions to enhance the security of the blockchain. In this survey chapter, we present a combination of common security issues and possible solutions. As in, the decentralized system does not depend on the trusted party but on cryptographic proof. [8].

1.2.1 Permissionless Vs Permissioned blockchains

Permissionless Blockchain is a public blockchain open for anyone to participate in, with no permission needed to gain access to the blockchain network. The identity remains unknown to the participants. Permissionless blockchain's main characteristics:

- Decentralized Permissionless blockchain architecture is fully decentralized, and there is no centralized unit that controls the network or is able to shut down the network.

- Anonymousness permissionless blockchain opens for everyone, resulting in anyone joining the network, and the participant can choose to stay relatively anonymous, as there is no need to declare the identity in order to get an address that allows performing transactions.

- The transparency for all the transactions is accessible by the public nodes and could also be defined as the data on permissionless blockchains being publicly visible.

- The performance depends on the gigantic number of participants in the network. To some extent, it is slower than the permissioned blockchain network, and the scalability is difficult.

- Consensus transaction cost is higher as the mining algorithm utilizes a lot of computing resources and electricity to solve the complicated mathematical equation. The consensus algorithm usually uses Proof-of-Work (PoW), Proof-of-Stake as we see that in cryptocurrency Bitcoin [17] [8] [18] and Ethereum [19].

A permissioned Blockchain is also called a private network that operates within a predefined set of known and identified participants. The following are some distinct characteristics:

- Fluctuating decentralized or partially decentralized since the concept of blockchain consensus validation or participation gives the network the nature of decentralized theory, which allows the admin to control who controls adding new nodes to the blockchain network.

- The consensus algorithm within the private network. The traditional algorithm should satisfy the validation criteria, which do not use and consume costly mining as Byzantine fault-tolerant (BFT), or crash fault-tolerant (CFT). Since

the participants in the network require permission to be granted, that offers more flexibility and gives the freedom of choice.

- Identified any participant needs to have permission to participate in the network. Based on the pre-determined access, the permissioned blockchain is also called a private network.

- The network performance is efficient. The transaction speed is faster, and the scalability is easy in the permissioned blockchain. The reason all the network members are predetermined includes a small set of participants. What makes the networks easy to scale is that the network participants are not an exception for unexpected growth since the access is controlled and not open to the public.

1.2.2 Hyperledger Fabric

Hyperledger Fabric is the most mature framework among the enterprise distributed ledger technology(DLT) platforms and operates as a permissioned network. The Hyperledger Fabric was established by IBM and then adopted by the Linux Foundation to become an open source. Then multiple technology providers joined a collaborative work, including IBM, Samsung, and Oracle, which had grown to over 60 organizations [20] providing an open-source enterprise-grade platform that operated and supported by the Linux Foundation [21]. Hyperledger Fabric is a framework involving a modular blockchain and configurable architecture, this includes key components such as consensus mechanisms and membership services. Fabric is a distributed operating system [22] [13] that enables network members to be recognized and must have permission granted in order to have access to the ledger. This type of Hyperledger is known as permissioned blockchain [13]. The benefit of the permissioned blockchain is to serve the business needs by maintaining the privacy and secrecy of the organization data [23]. One of the intriguing features Fabric brings is the ability to create subnet networks within the organization called channels. The network channels maintain

access levels for the network's members for isolating and managing peers' access to the resources by either grouping peers or an organization into an isolated subnet within the main network. The architecture of the Fabric network emphasizes the confidentiality of the resources and secure access control.

1.2.3 Hyperledger Fabric Architecture

The fabric architecture facilitates a distributed system that is composed of multiple nodes connected to form a permissioned blockchain. Then, the permissioned blockchain Hyperledger Fabric provides numerous APIs for creating chaincode (smart contracts) in a variety of computer languages, including Go, Node.js, or Java. Fabric goal is to record transactions activity in an immutable ledger.

In Figure 1.1, the data life-cycle with Fabric framework components shows the sequence of the operation, starting with the client submitting the request and ending with updating the ledger in every single node [24]. The blockchain in the Fabric framework running software contains the business logic called a **smart contract**, commonly referred to as **chaincode** within the Hyperledger Fabric framework, defines the business logic that governs transactions and interactions between participants in the network [24]. The chaincode stores the ledger and the state date for the operation task. The chaincode is also responsible for executing the transactions. These transactions will not have an impact on the ledger state until they become *"endorsed"* since only transactions that were endorsed are allowed to be committed on the state. Hyperledger Fabric allows for the existence of one or more specialized chaincodes designed for management functions and configuration parameters.The term "system chaincode" encompasses all chaincodes specifically designed for management purposes within the Hyperledger Fabric framework. These chaincodes play a crucial role in overseeing and regulating various aspects of the network, ensuring efficient governance and operational integrity. [24]. Each component has a specific role to achieve different purposes. The transaction data flow could be represented in four

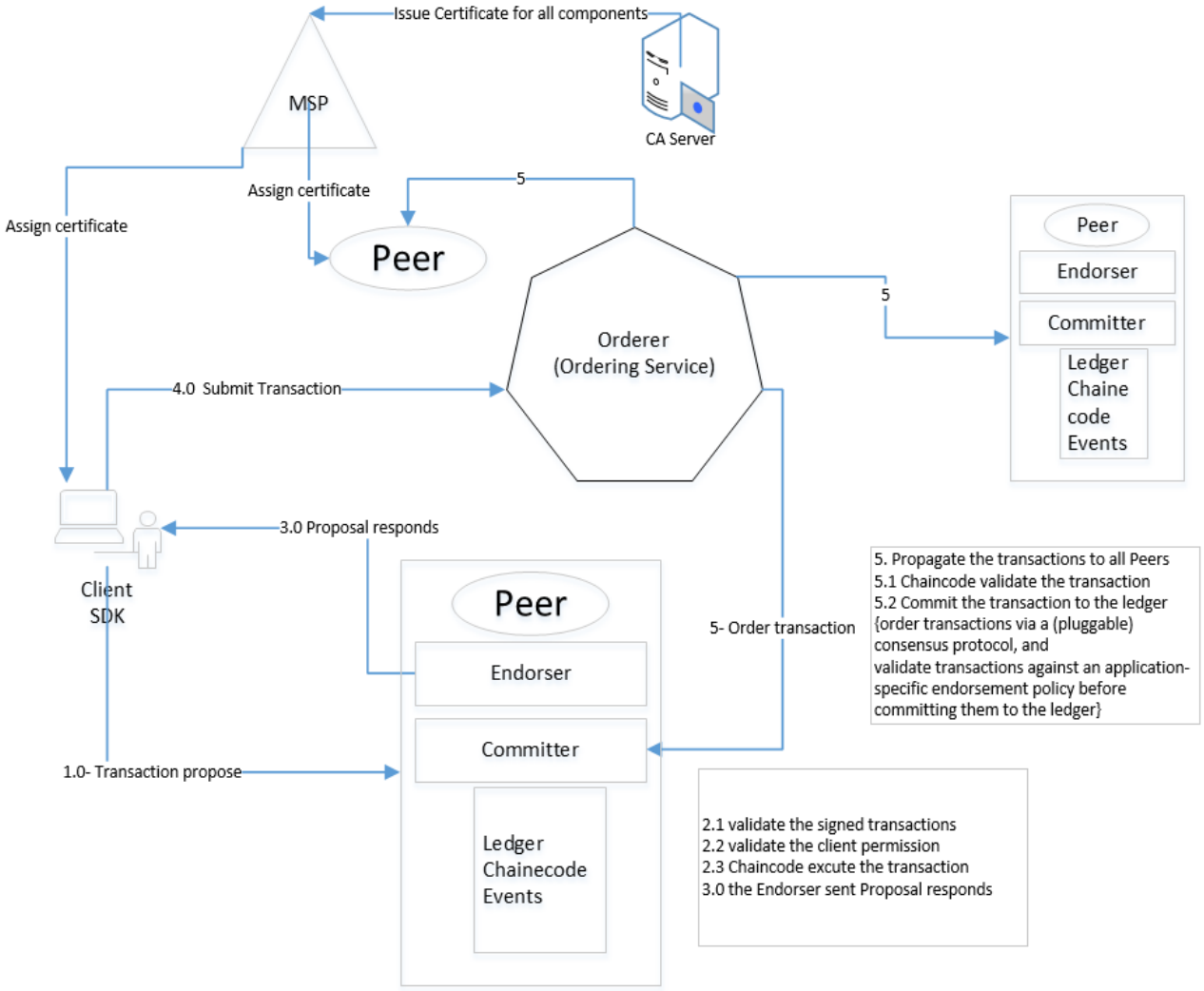


Figure 1.1: Hyperledger Fabric Architecture

main phases: *endorsement*, *ordering*, *validation* and *committing*.

1.2.4 Hyperledger Fabric components

- **Assets** is anything with monetary value or information that is considered an asset. Each asset contains two components: ownership and state [25]. In Hyperledger Fabric, assets are represented as a set of key-value pairs.

- **Peer Nodes** is hosting and storing the smart contracts and the ledgers, which made it a critical and major component of the Fabric network. The peer node maintains the transaction log and the State. The log is immutable once it's

added, it cannot be changed or deleted, state database and the chaincode deployed to it. The peer node exposes a service, and those services are built on a Google Remote procedure call (gRPC). The services are available to be invoked by clients, peers, and by the Orderers. The Order service uses the service to send a block to the peer. Peers exchange block data by using gossip [26] data communication protocol by utilizing randomized block swaps between peers, which gives the ordering nodes the operation and the responsibility to disseminate newer transaction blocks consistency to all network peers, as introduced in [26]. The performance of the gossip can be enhanced up to 10 times as presented in ICDCS 2021 conference [26], As shown in Fig. 2.0.

- **Ledger** is the component that maintains the state and ownership of an asset recorded in the ledger. The ledger is composed of two parts:

    1. **World state** It is the ledger's database. It describes the ledger's state at a specific point in time. To provide direct access to the current or latest value of a state, for each valid transaction, the peers update the ledger world state by committing the most recent values as key-value pairs. The fabric default database is LevelDB [27], which is used for the world state [28]. Also commonly uses CouchDB [29].

    2. **Blockchain** serves as a comprehensive transaction history log that meticulously tracks every transaction that occurs within the network. In contrast to traditional databases, the data recorded on the blockchain is unchangeable, often referred to as immutable, meaning it cannot be altered or modified once it has been entered into the system [28]. This characteristic ensures the integrity and authenticity of the transaction history, making blockchain a trusted solution for maintaining secure records in various applications.

- **Smart contract** is an application that holds the business logic for the blockchain systems. In the Hyperledger Fabric framework, it's referred to as **chaincode**. It is the software responsible for the assets and the transactions of the assets. it uses the ledger to interact with [25]. The smart contract operates as a dependable distributed application along with the blockchain. The blockchain grants the chaincode security, trust, and the underlying peers' consensus to guarantee a secured system. One of the essential factors to mention when it comes to smart contracts is that the application code is considered vulnerable. The majority of blockchain platforms that support smart contracts currently use The **order-execute** architecture signifies that the consensus protocol is designed to first arrange transactions in a specific order before they are executed. This approach is crucial for ensuring that all participants in the network agree on the sequence of transactions, thereby maintaining the integrity and consistency of the blockchain. (1) Orders the transaction, validates it, and finally broadcasts the transaction to all other peer nodes, (2) The transactions are executed progressively by each peer after another. A new architecture approach that Fabric introduced is called **execute-order-validate** (EOV). breaking down the transaction process into three stages claimed it solved the scalability, rendering, flexibility, and confidentiality that the order-execute has to deal with a lot of those issues, (1) *execute* Carry out a transaction and ensure that it is accurate to endorse it, (2) *order* is a (pluggable) consensus mechanism to arrange transactions, (3) *validate,* Before committing transactions to the ledger, a verification trigger in opposition to an endorsement policy that is specific to the application [28].

- **Channel** is a type of logical organization to group a set of peers. Channel has the authority to access an independent ledger that can belong to multiple organizations. Fabric defines a *channel* as a private communication subnet between

two or more specific network members. A channel is defined by members of one or more organizations, ordering service nodes (OSNs), shared ledgers, and chaincode applications [28].

- **An organization** is a set of peers representing a department or agency sharing their particular resources with the collective network, which makes it possible for the Fabric network to exist. The Fabric's network fundamental hosting peers who participate in the network are members of different organizations. The organization's Membership Service Provider assigns a digital certificate to give identity to each peer member [28].

- **Membership Services Provider (MSP)** manages Hyperledger Fabric network participants IDs and authentications. The MSP implemented a Certificate Authority(CA) to keep track of the certificates that are used to verify a member's identity and roles. There is no unknown identity that can participate in the Hyperledger network, as it is a permission network or private [28].

- **Ordering Service** Another name for it is an *"ordering node"* and *Orderers* [28]. The Ordering service implements the transaction's order, and the primary responsibility of the Orderer node is to ensure a consistent state of the ledger across the Hyperledger Fabric network [28]. Ordering Service provides a consensus mechanism and makes sure that the order of the transactions is maintained. Interacting with peers and endorsing peers ensures the network transaction delivery [28]. The transactions are packaged into blocks and transmitted to peers through a channel. The previously supported configuration messaging systems for the Ordering Service are Solo (deprecated in v2.x), it is composed of just one ordering node, and Kafka crash fault-tolerant (CFT) solution that makes use of leader and followers nodes also deprecated in Fabric version 2.x. Raft is the recommended protocol option by Hyperledger as Raft is a crash fault-tolerant

(CFT) ordering service that is new as of version 1.4.1 [28]. according to Raft implementation [30] by *etcd* library [31]. The list of organizations that are permitted to build channels is also maintained by the ordering service. The term "consortium" refers to a group of organizations. The list of the organization's group is stored in the ordering system channel configuration. The basic channel access control is also enforced by Orderers. Hyperledger's consensus type is deterministic consensus algorithms, this implies that any block that has been peer-validated is guaranteed to be final and accurate.

1.2.5 Digital signature

It is a method based on mathematics that verifies a message's integrity and validity by devising a construct that verifies both the source and the content of the message in a manner that can be demonstrated to an impartial third party [32]. The blockchain network consists of three main components: the Smart contract, the ledger, and the ordering node. However, there is one more crucial component that holds everything together, which is the cryptography and, to be more particular, the digital signatures [33]. With the digital signature, we achieved confidentiality and authentication. The recently reported security vulnerability of Hyperledger Fabric was recorded in July 2021 regarding the RSA security issue in section 5.5, we list a survey for the different signature models or schemes. Starting In 1984, Shamir [34] introduced identity-based encryption, aiming to simplify certificate management. Since that time, many schemes were proposed but kept the issue open till 2001 when Boneh and Franklin [35] proposed an identity-based encryption scheme that is fully functioning (IBE).

In the following paragraph, we discuss several digital signature schemes in the blockchain network.

1.2.5.1 Single Signature

It is the default mode for most implementations as a default approach, which is still used for most transactions today. One approver uses a private key to create one signature on a transaction record. Regardless of the fact that the implementation is simple and low-cost, The Single Signature has the lowest levels of security because if the holder of the key gets hacked, their funds are effectively gone.

1.2.5.2 Multi-signature

It is an alternative scheme that introduces the idea of having multiple approvers, each with their own private key to generate their own signature where each approver signature is recorded as part of the transaction record [36]. Multi-signature is exponentially more secure than the standard single-party approval scheme. However, it does introduce some undesirable attributes, primarily associated with the recording of each signature of the approvers as part of the transaction record.

1.2.5.3 Threshold signatures

are widely embraced in blockchain applications. It enables approvers to co-operatively share a secret known as a private key with other participants in the blockchain network, disregarding the need for a Certificate Authority(CA). Then, the signature shares are all combined to verify and reveal any single transaction. One of the prominent digital signature schemes is called the Elliptic Curve Digital Signature Algorithm(ECDSA) [37], which is the standard signature scheme used for the Bitcoin transaction system. The ECDSA scheme runs four distinct algorithms *Setup*, *keygenerattion* and *Signaturegeneration* and lastly, $Verification$. To illustrate, ECDSA requires several approvers to

combine their signatures to have control over a blockchain transaction jointly. ECDSA also uses a technique called multi-party computation(MPC). The technique generates one key share for each approver. The keys are never reconstructed or tampered with. The multi-party approval scheme allows at least M of n shares to be available to generate the ECDSA signatures. The threshold cryptographic scheme could suffer from having a corrupted approver or potential peer collusion, though the ability to establish and distribute lost key shares and Devising the signature to append a new transaction in the order of milliseconds is a great security enhancement. Nevertheless, ECDSA Scheme lacks dealing with a forged certificate and the ability to delete or drop off a malicious participant. It is important to note that recent surveys [38], [39] discussed in detail the threshold signature schemes.

1.2.6 Transaction Flow

Transactional mechanisms [40] that occur during a typical asset trade or exchange. Assuming a real-life scenario for two clients **A** and **B**, each client has its own peer access to interact with the network in order to communicate by sending and querying the transactions to and from the ledger. In order to clarify the process, the steps are the following:

1. The client initiates a transaction.
2. Endorsing peers validate the signature and execute the transaction.
3. The proposal responses are reviewed.
4. The client compiles the endorsements into a complete transaction.
5. The transaction undergoes validation and is then committed.
6. The ledger is updated accordingly.

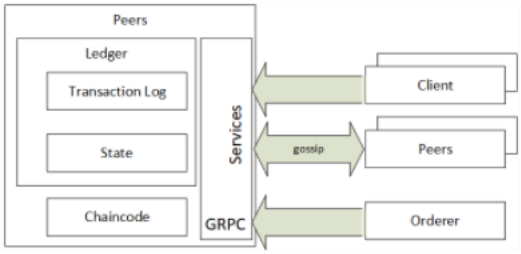


Figure 1.2: Peer node architecture and communication nodes

1.3 Consensus

For fault-tolerant distributed systems, consensus is a fundamental model. Each transaction proposes a value to the other nodes. All valid transactions processed by different nodes must reach a consensus on the same value, which should match the originally proposed values [41]. The blockchain technology is released on quorum-based consensus in order to commit the transaction. The number of valid peers varies from one consensus algorithm to another. *Quorum* represents the minimum number of votes needed from node peers for a distributed transaction to gain authorization to execute an operation in a distributed system. The quorum is designed to ensure that a distributed system operates consistently. The Hyperledger Fabric utilizes Raft as an order service algorithm that requires half of the nodes to verify and endorse a new block. The formula is represented as $Q = \frac{1}{2}n$, (Q is the quorum, n is the total node). Hyperledger Fabric is a private blockchain. Therefore, the consensus mechanism requires a central authority or group of endorsers to validate and approve updating the ledger by writing the transaction in the sequence of each new transaction. The main goal of the consensus process is to decide on the validity and authenticity of the new block while also checking the consistency of every replica and then synchronizing them throughout the network peers. One of the popular protocols that is discussed in the literature is called Crash Fault Tolerance(CFT), similar to Raft. It suffers from malicious attacks and faults since only half of the nodes are needed to validate new blocks. On the other hand, Practical Byzantine Fault Tolerance (PBFT) requires

two-thirds of the nodes to endorse a new block to be added to the ledger. *Block Height* is the number of a specific block location of a transaction that has been completed in the past in the blockchain or the present size of the blockchain. As the block height identifies the number of the blocks currently proceeding in the blockchain, The majority of consensus algorithms work by agreeing to mine the chain with the largest block height. The **Genesis block**, like other blocks in the network, is the inaugural block in a blockchain and serves as the foundational reference point. There are no blocks to proceed in the blockchain, it has a block height of zero.

1.3.1 CFT Crash fault-tolerance

Crash fault-tolerance (CFT) is the de-facto consensus algorithm for the Hyperledger Fabric, its mandates half of the node to validate and confirm new transactions. Raft [42] was built as an enhancement of what Leslie Lamport introduced the Paxos [43] [44] to provide a convenient way for students to learn about consensus than Paxos [30]. *Paxos* is the primary research material used for students teaching about consensus. Two commonly used consensus protocol frameworks that have been used for Hyperledger Fabric are Apache Kafka and Raft. In the next subsection, we thoroughly describe each consensus protocol:

1.3.1.1 Apache Kafka

Beginning with its initial release, the ordering service was developed utilizing the Apache Kafka messaging system. Apache Kafka is designed with crash fault tolerance (CFT) and operates on a leader-follower node configuration. To begin with, the client sends transactions to the ordering service nodes (OSNs) to submit transactions to the Kafka topic as one topic per channel and vice versa, the ordering service node will consume from the Kafka topics. Kafka has now been deprecated from Hyperledger Fabric since release 2.0, and for this reason, we are not surveying the system in this chapter in detail.

1.3.1.2 Raft

[30] It is a new consensus protocol for Hyperledger Fabric that started to be used as version 1.4.1. Raft is a crash fault-tolerant (CFT) ordering service [28] based on a Raft protocol implementation [30] in *etcd* library [31]. Raft imposes a higher level of coherence in order to decrease the number of states that must be taken into account to validate the transactions. As mentioned above, Raft [42] was built as an alternative to **Paxos** by Leslie Lamport [43] [44] to provide a convenient way for students to learn about consensus than **Paxos** [30] [45]. **Paxos** was the primary research material used for students teaching about consensus. There are several problems with RAFT:

1. RAFT consensus is not a Byzantine fault tolerance algorithm.
2. The client setting in RAFT is only required to submit a single order with a transaction proposal.
3. The followers nodes to the Raft Ordering service nodes trust the leader's block proposal without validation as it can be a "crash fail".
4. In Raft consensus, the transaction reaches consensus then delivered to the node to be signed prior to being saved to the block store.

1.3.2 BFT

The Byzantine Fault Tolerance (BFT) consensus mechanism is a fundamental approach used in distributed systems to achieve agreement among nodes despite the presence of Byzantine faults, which can manifest as arbitrary or malicious behavior by a subset of participants. BFT consensus protocols ensure that the network can reach a consistent decision even if some nodes fail or act in a Byzantine manner.

BFT consensus mechanisms are essential for ensuring the security and integrity of distributed ledger systems, particularly in settings where trust among participants

cannot be assumed. BFT consensus plays a crucial role in enabling decentralized networks to function reliably in the presence of adversarial actors.

1.3.2.1 PBFT

It is called Practical Byzantine Fault Tolerance (PBFT). In private and permissioned blockchains, PBFT is widely utilized. In PBFT-based blockchain systems, the network architecture consists of a set of active and inactive participants or nodes [46]. Within the live node, a primary one is designated to handle incoming messages from clients and subsequently disseminate them to the other nodes [46]. This system exhibits a more robust trust model in comparison to PoS and PoW. Similarly, the PBFT provides a solution that solves the issue of the Byzantine Generals Problem in [47], Lamport, L., Shostak, R., & Pease, M indicated that no solution of presents of $\frac{1}{3}$ or greater potential malicious nodes as the potential traitor generals called Byzantine node. The PBFT is also discussed in [48] [49] [50], in 1999 by Miguel Castro and Berbara Liskov [50] proposed a paper with a solution for the Byzantine generals problem, to solve the Byzantine faults in a network or a system includes:

$$n = 3f + 1$$

n represents the total number of valid nodes and consider f is the maximum expected malicious nodes. PBFT is commonly used for the permissioned blockchains. PBFT is a fundamental concept that has been included in the design of numerous commonly used BFT systems. [51] The BFT failure could be categorized into two sections:

1. Fail-stop: nodes can crash, no values to be returned from that node.

2. Byzantine node: nodes can send malicious values that are incorrect or manipulated.

The PBFT protocol suffers from some deficiencies elaborated in [52]:

1. PBFT consensus protocol standard mechanism allows the system to process the client requests one by one at a time in order. Still, when the system exposes numerous requests, the protocol performance will drop and become inefficient.

2. Assuming N is the total nodes, PBFT requires delivering messages:

$$N = 2N^2 + N$$

   times to achieve consensus, and it will become a high cost for message transmission.

3. Master node selection in PBFT consensus relies on round-robin as it considers a simple mechanism, and the location of the master node will be easily identified by a malicious attack, which exposes the system to the attack.

1.3.2.2 Tendermint BFT protocol

Tendermint protocol [53] is one of the consensus algorithms that the Hyperledger team considered to be used in Fabric [54] as it is founded on PBFT protocol. To achieve consensus, the system nodes need to be three times grater than the expected total malicious nodes, plus one $n = 3t + 1$. The Tendermint platform open source is available in [55]. In this section, we aim to present a literature review on how the Tendermint works. The protocol at runtime has five variables maintained by each node or participant (step, (2-locked V&R) lockedV, lockedR, (2-valid V&R) validV, and ValidR) [51]. To achieve consensus at height h, the protocol operates in a round-robin manner, transitioning from one round to the next. [51]. Those steps *propose*, *prevote*, and *pre-commit* are required in each round. [51] introduce three types of attacks that will transit the state of the consensus to deadlock before GST as well after GST, which basically means no more blocks that could be added to the blockchain. That type of attack is categorized as DoS as the most important property,

which is the liveness that cannot be achieved. In other words, the consensus cannot be achieved.

1.3.2.3 BFT-SMART consensus

Another interesting feature of PBFT is its compelling design as a state machine replication (SMR) for the distributed consensus systems [56]. It was first introduced in 2007 to devise effectively a BFT total order. BFT-SMART adopts the commune system model for BFT $n \geq 3f + 1$ nodes, where n is the number of nodes and f represents the Byzantine faults. BFT-SMART gave the ability to change n and f at runtime over *join* and *leave* operations, which allow the consensus protocol to be configured to be only $n \geq 2f + 1$ replicas to tolerate f crash faults [56].

Regardless of the configuration, for communication, the distributed systems require steady point-to-point connection links among processes. the connections are implemented over TCP/IP using Message Authentication Codes (MACs). The replica-to-replica channels using symmetric keys are generated throughout Signed Diffie-Hellman, each replica utilizing a pair of RSA keys [56]. The client-replica channel keys are produced using the endpoint IDs, and the clients do not require storing or holding the key pairs [56]. Nevertheless, the mechanism of the BFT-SMART consensus has a presumption of non-malicious non-malicious Byzantine errors exist [56].

At the latest attempt in 2018 by Joao Sousa and Alysson Bessani and Marko Vukolic in [12] [57], The authors investigated Fabric with a BFT ordering service, then modified the Apache Kafka-based ordering service and replaced Kafka with a cluster of BFT-SMART servers [54]. The Fabric community did not accept that attempt [58]. That decision regarding the rejection was made on several reasons will be elaborated in future sections. The primary reason was the PBFT consensus tool was not built as a stand-alone ordering service, but it was built on top of BFT mechanisms that were already in place [12]. It was literally built based on Apache Kafka infrastructure.

1.3.3 Networks and Synchrony

When the message is propagated over the network, it uses different models to guarantee the liveness property. In previous work [59], the author discussed synchronous networks, indicating the duration required to transmit a message between nodes is constrained by a predetermined *fixed upper bound*. However, In an asynchronous network, there are no established fixed upper limits. The Hyperledger Fabric consensus protocol, Raft, operates within a partially synchronous network model. Another decentralized consensus model is called Partially Synchronous Network. This concept is known in a distributed system where the upper bound is unknown, which is the time needed for a message that node N1 sent to another node N2. It is important to mention that the time period cannot be infinite. The upper bound is different by $\triangle$, the time required for a message to be sent. Where $\triangle < \infty$. The message can be received by delta. In [51] [59]considered two types of partial synchronous networks

Type I: The upper bound, or basically the delivery estimate time before timeout, is unknown to the participants, and it has to hold a positive integer value less than infinite. the total estimated delivery time = $\triangle$.

Type II: The amount of the upper bound $\triangle$ is known to the participant, but the protocol designer adds an unknown period of time, called Global Stabilization Time (GST) [51] triggers first then triggers the counter for the known fix $\triangle$ when the upper bound it has to be a positive integer value less than infinite, the total estimated delivery time = $GST + \triangle$

where the GST is unknown and the $\triangle < \infty$ is known. In `Type I`, It was impossible to overlook the message with Denial of Service (DoS) attacks. The delivery of all messages is assured. Therefore, the DoS is not possible or not allowed to occur on a system that uses the Type I partially synchronous network [51]. In `Type II`, there are two time periods. The first one is an unknown GST, and the second one is known as $\triangle$. After the unknown period GST, the model will transfer from partial synchronous

networks to synchronous networks. Therefore, the DoS is not allowed. Still, the issue becomes when the attacks happen before the unknown time periods or before the GST that will cause a DoS that cannot be removed even after the GST [51].

1.4 Attacks and Security issue

The below addresses a couple of research papers to discuss the security issues collected from several papers. The next section presents a literature review on the common open issues that each one is considered a research area.

1.4.1 Cryptographic Weakness. Invalid RSA PKCS signatures

By reviewing the Jira dashboard of Hyperledger Fabric, a new critical vulnerability has been reported in Fabric *Jira board* on July-14-2021 also recorded into the National Vulnerability Database (NVD) CVE-2021-30246 [60] some invalid RSA PKCS# 1 v1.5 signatures are mistakenly recognized to be valid. This vulnerability has been handled as CVE-2021-30246 since 04/07/2021. The exploitation of that type of attack is known to be difficult. The attack needs to be done within the local network. This is possible for the Hyperledger Fabric since they have the authority to participate in the network and submit valid transactions.

1.4.2 Impact of Network Latency on Hyperledger Fabric

Thanh Son, Guillaume Jourjon, Maria Potop-Butucaru and Kim Loan Thai [61] experiment the Hyperledger Fabric network performance, finding a huge network delay after examining a wide network setup for the Hyperledger Fabric between Germany and France however by monitoring the delay of the network over time increases significantly average 3.5 seconds among two clouds systems. In the paper, the researchers noticed that approximately 134 seconds offset when they complete adding 100 blocks to the blockchain from one cloud to another or the time it takes to deliver the last block to the other peer. This is a significant proof of consistency issues that critical businesses can suffer, such as bank or exchange business. That impact caused network

delays on a PBFT based blockchain. In their paper, going over the main components of Hyperledger Fabric and the Execute-Order-Validate architecture go into the details of the limitations of Order-Execute architecture.

1.4.3 Malleability Attack

This attack started with Bitcoin. In February 2014 MtGox, The once-largest Bitcoin exchange, shut down and filed for bankruptcy, saying that attackers drained its accounts using malleability attacks [62]. This incident was Bitcoin's $460 Million Disaster [63]. The attacker gains the ability to mount a malleability attack, which gives him the ability to intercept, modify, and rebroadcast a transaction. When that happens, the transaction issuer will not receive the confirmation and believe that the original transaction was not sent successfully [62]. The logic behind this attack is that the signatures that contain the ownership of bitcoins being transferred in a transaction are not immutable, and it can be modified and do not provide any integrity guarantee [62]. In Hyperledger Fabric [49], when the Orderer broadcasts the transactions to the peers, the attacker within the network can modify the transaction information, modify the receiver identity, and generate a new transaction hash to be rebroadcast again. At this point, the original transaction by the client who submitted it will be on hold since the client is waiting for a confirmation response for the submitted transaction by the endorsement, which will never happen since the adversary has already modified the transaction hash [64]. The timeout will trigger a resent activity for the lost transaction.

1.4.4 Wormhole Attack

The peer that is part of the private Hyperledger Fabric network can create a virtual private network with another network outside the Fabric for leaking information of the network that the peer is a member of its private network. The wormhole attack appeared only within a private network. The attacker is already getting access and

creating a virtual private network within his peer network, which could be his laptop. This laptop is connected to the company network from its private network with the outside network and leaked information [64]. This attack does not require any knowledge of honest members to be launched on the Fabric private network [64].

1.4.5 Private Key Leakage

In the Hyperledger Fabric, each participant is responsible for maintaining the secrecy of their private keys. It is possible that an external malicious entity gets hold of the private keys (in case it is stored in a container or a compromised host). The implications of this leak cause further attacks such as message tampering and man-in-the-middle attacks, which allow malicious users to obtain more credentials, infer sufficient information about security events, and cause insider threat in the Peer to Peer Network [65].

1.4.6 Third Party Digital Certificate

In Hyperledger Fabric, the service provider relies on both the certificate authority (CA) and the identity manager to securely distribute cryptographic keys to participants within a peer-to-peer (P2P) network. The role of the CA is critical, as it is responsible for issuing certificates that authenticate the identities of network participants, ensuring that only legitimate entities can join and interact with the blockchain network. The service provider, in turn, trusts that the certificates issued by the CA are valid and reliable. However, recent research has highlighted potential vulnerabilities in this trust model. Specifically, it has been demonstrated that a third-party entity, such as the CA, can be compromised, leading to the generation of fraudulent certificates and fake identities among P2P nodes. This presents a significant security risk to the overall integrity of the network, as malicious actors could exploit these fake identities to carry out attacks or unauthorized transactions, undermining the trust and reliability of the blockchain infrastructure.

1.4.7 Censorship Attack

Censorship attack is a variant of $> 50\%$ *or 51% Attack* [66]. This type of attack in prevailing blockchains is ubiquitous. [67] There are different types of censorship attacks in consensus algorithms [66]. The common definition of the censorship attack is defined as a *"majority attacking" mechanism* to establish a chain that denies the transactions that the regular validator, client, or minor would approve [66]. At the time of this survey, there was no efficient solution. The attack occurred in permissioned and permissionless blockchains. Recent incidents targeting the permissionless blockchain, including Ethereum and Bitcoin. Yet it threatens the permissioned blockchain indicated in recent research [66] introduced two methods to enhance the consensus protocol mechanism in partial synchronize network for PoW and PoS algorithms, that explained in detail regarding the proposed solutions in the related work section. The attack happens when a single ordering node controls a group of nodes to maliciously impact the majority of the node's decision to validate and propagate the new transaction. One variant of this attack is called censorship attack, proposed by Vitalik et al [68].

1.4.8 Hijacking Attack

This attack occurs at the networking level. The attacker seizes control of the communication. It is considered an attack on the network security. In [25], the researchers expose two hijacking methods in their paper. The one that could be a threat to the Hyperledger Fabric is the delaying propagation. The authors exhibited that "any network attacker can hijack few $(< 100) BGP$ prefixes to isolate 50% off the mining power" and "slow down block propagation by interfering with few key Bitcoin messages" [61]. Due to the fact that the full-fledged attack type on the Ethereum blockchain is based on the hijacking assault to burglarize cryptocurrency [25]. The researcher group in [61] considers hijacking PBFT based blockchains. Perform a

hijack attack targeting Hyperledger Fabric by postponing the spread of blocks [61]. Remarkably, the permissioned blockchain behavior of Hyperledger Fabric does not prevent this type of attack, and it could be exposed to it easily [61].

1.5 Related Work

1.5.1 Using BFT-SMART consensus protocol

As the latest attempt to update the consensus by Joao Sousa and Alysson Bessani and Marko Vukolic in 2018 [12], in their most recent attempt to offer Fabric with a BFT ordering service, modified Fabric v1.1. Kafka-based ordering service and replaced Kafka with a cluster of BFT-SMART servers. The Fabric community did not accept that attempt [58]. That decision regarding the rejection was made on several reasons will be elaborated in future sections. The primary reason was the PBFT consensus tool was not built independently, but it was designed on top of existing BFT systems. It was literally built based on Apache Kafka infrastructure. Proposing the work. After we investigated, we don't see core enhancement of the previous implementation in 2017 [57].

1.5.2 Censorship Attack Protection

The existing implementation of Hyperledger Fabric client SDK sends the proposed transactions to one Orderer service within the organization, which gives the single organization admin the ability to manipulate the validator rule. This type of attack disseminates dishonest transactions while the other honest validators will not be able to verify. Another research in [66] proposed two methods to improve consensus mechanisms against censorship attacks. The first technique adds three types of messages and an auxiliary role for network nodes to the Tendermint consensus process. It can identify attackers and automatically organize the approved blocks by honest validators into a trustworthy chain. The Second method is using a network that must manually switch to a blockchain established by more trustworthy validators compared

to a suspicious percentage of censorship attacks. The related work has never been researched or implemented on permissioned blockchain. Despite the fact that the second technique is not automated, it is still more efficient than relying on the market to select the honest chain. The second method was based on the Casper FFG consensus mechanism by applying the same theory, making the client JDK submit the proposed transaction to all ordering service nodes for validation.

1.5.3 Hiding Leader Node Identity

Nitish Andola, Raghav, Manas Gogoi and S. Venkatesan, Shekhar Verma release a research paper with zero knowledge in [64] indicating two security limitations of Hyperledger Fabric and introduce a possible solution. First, when an endorser's identity is made public to all members of a channel, it becomes vulnerable to denial-of-service (DoS) attacks directed at blocking client-related operations or undermining network performance. In next section of this paper I added more in-depth details about the attacks that caused the DoS. The key reason for the denial of service (DoS) is that the endorser's identity is known to all peers. The authors in [64] proposes two solutions to solve this attack, by hiding the leader node identity since the nodes are available to the public by using Verifiable Random Function (VRF), or pseudonyms leader identity. Either of the presented solutions improves the network performance and efficiency, reducing the network throughput with the effects of response time that took longer. Second, in Hyperledger Fabric, the peer's identity is known to other peers within the channel, making the system susceptible to wormhole attacks. Leakage of its ledger information occurs when a peer member is compromised outside the channel, resulting in exposing all peer's info since the ledger is the same for all network members. The author propose this function as verifiable random with a primary process is to choseover a random endorsers. In the second proposal, the solution uses pseudonyms to anonymize the endorsers. Also, inside the channel, the identity of either the sender or receiver has been anonymized. The researcher used a group set of

a signature approach. An anonymous method that uses bilinear pairing to create zero knowledge about the recipient [64]. The sender's identity is anonymized by a bilinear pairing [64]. With that approach, the network gains protection from the malleability attack as well. In [64], researchers provide proof for Signature Unforgeability and Unlinkability in Ciphertext (UN-C). The results were provided by measuring the effects of the DoS attack on the Hyperledger Fabric network. The team works on recording the transaction rate without DoS on two peers with a transaction rate of 100 TPS at a send rate of 123 TPS. After applying the DoS, it causes a lack of performance to drop to 100 TPS. The send rate response time has increased from 1.396 s to 2.44 s [64].

1.6 Future Work

One of the significant findings to emerge from this study is that we need to replace the RAFT consensus protocol with another protocol that leverages the predecessor DLS algorithm with a successor algorithm called BDLS protocol [51] that was introduced by Dr. Wang. Provide a proof of concept regarding the well integration with the Hyperledger Fabric regarding the security concerns, especially against the DoS attack. The aim is to provide end-to-end stand-alone BDLS protocol consensus.

1.6.0.1 Replacing RAFT with BDLS

In order to solve the security issue of the partially synchronous network in Crash Fault Tolerance (CFT), we replace the Raft consensus protocol with a BDLS consensus based on the DLS protocol algorithm.

This study has raised important questions about the nature of the Malleability Attack in section 1.4.3 since the keys are not available for the attacker in order to be replaced as the technology has been used in BDLS is a Threshold signature scheme. I aim to provide approval on this area in future research.

1.6.0.2 Replacing SDK

The Second goal for the future direction is to provide an end-to-end network system that utilizes PBFT as a consensus protocol. We plan to design and build a new client Software development kit (SDK) for Hyperledger Fabric to disseminate the proposed transactions to all Orderer nodes. This can be achieved by establishing a connection to the Orderer Services, which contains the information for the other Orderer nodes, with the network. The decentralized network includes mutable organizations. to force all the organization's validators nodes to validate the transactions, which provides a security layer to the Hyperledger Fabric network against the censorship attack in section 1.4.7.

1.6.0.3 Performance Speed assessment

Although this study focuses on Hyperledger Fabric transactions speed. In other words, setting an experimental environment to collect data to analyze the transaction throughput is also called the transactions per second(TPS). This study has a bearing on BDLS participants with linear communication/authenticator complexity who could reach an agreement in 4 steps [51]. On the other hand, at least seven steps are involved in the best linear communication/authenticator complexity methods currently in use to reach consensus [51].

1.6.0.4 New Platform

In the upcoming chapters, we will demonstrate how to develop a new platform for the Hyperledger Fabric framework for the network. This platform will utilize the BDLS consensus protocol to run the ordering service node (OSN). Furthermore, we will measure the performance speed throughput.

1.7 Conclusions

In conclusion, this literature review chapter, we presents several security challenges in Hyperledger Fabric consensus protocols and discusses relevant attacks that impact

the safety of the consensus protocols to approve and append new transactions to the ledger. Undoubtedly, this research domain is fast-developing and offers future opportunities to investigate and contribute to developing more secure consensus algorithms and practical decentralized blockchain systems. In general, decentralized applications bring innovative ways to store, update, and read data. However, the service's blockchain provides can be hindered due to the security flaws that exist in blockchain consensus algorithms. In this chapter, we discussed Hyperledger Fabric's basic concepts and the relevant security attacks that impact decentralized blockchain applications. We also comparatively studied each consensus algorithm, highlighting its strengths and weaknesses. We aim to extend our work in BDLS protocol implementation to replace RAFT protocol in Hyperledger Fabric with end-to-end BFT secure tamper-proof decentralized consensus ordering services.

CHAPTER 2: Hyperledger Fabric Orderer

This material is reprinted with permission from Ahmed Al Salih, "Pluggable Consensus in Hyperledger Fabric," Proceedings of the 2024 6th Blockchain and Internet of Things Conference (BIOTC 2024), ACM, ©2024. DOI: 10.1145/3688225.3688237

2.1 Introduction

The Hyperledger Fabric Orderer system is integral to the consistency and reliability of transactions within a Hyperledger Fabric network or distributed nodes. The selection of a relevant consensus protocol is essential to achieving these objectives. This research delves into the technical aspects of implementing the Hyperledger Fabric Orderer system, focusing on the Raft consensus protocol. Furthermore, it explores the steps required to plug any consensus protocol into the Hyperledger Fabric Orderer service, exemplifying this process by integrating the BDLS protocol [5]. The Raft protocol [6], known for its simplicity and Crash fault-tolerance characteristics, has gained prominence in distributed systems. By understanding its inner workings, participants can grasp how it enhances the resiliency and performance of the Orderer system. Additionally, exploring the steps required to integrate the BDLS protocol is a foundation for seamlessly incorporating other consensus protocols. This research will go beyond theoretical concepts and provide practical insights into technical implementation. We will share our firsthand experience integrating the BDLS consensus protocol, proposing precise steps for the integration phases. By showcasing this integration, a real production-ready example provides a deeper understanding of the intricacies and advantages of combining alternative consensus protocols with the Hyperledger Fabric Orderer system. Overall, this research aims to provide a

comprehensive knowledge of the technical implementation of the Hyperledger Fabric Orderer system. The chapter focuses on the last supported consensus protocol, the ETCD Raft protocol. Indicating the integration of the new BFT protocol, the BDLS protocol, guarantees practical insights into the steps required to plug any consensus protocol into the Hyperledger Fabric Orderer service (OSN).

2.2 Fabric Orderer

The Orderer, also known as the ordering node service (OSN) in Hyperledger Fabric [13], is the main component responsible for ordering and validating transactions before committing to the blockchain. It guarantees that every node in the network has an identical perspective of the transaction order by offering a distributed ordering service. The Orderer maintains the ledger's integrity and guarantees that transactions are not tampered with or duplicated.

The Hyperledger Fabric's Orderer uses a consensus mechanism to order transactions. There are several consensus mechanisms available, including Solo [13], Kafka [69], and Raft [6] as of Fabric's latest release 2.5 [70]. Solo is a simple consensus mechanism that is suitable for testing and development environments, while Kafka and Raft are robust and suitable for production environments.

The Orderer in Hyperledger Fabric architecture is supposed to be modular and pluggable [11]. Therefore, organizations can choose the consensus mechanism that best serves operation needs. Additionally, the Orderer can be run as a standalone service or as a part of a more extensive network.

In the Hyperledger Fabric's network, the client submits a transaction through an individual node. Therefore, the node that receives and validates the transaction then submits the transaction to the Orderer node. The Orderer receives these transactions, assembles them into blocks, and guarantees to achieve consensus among all Orderer nodes. After achieving consensus on a block by all Orderer's nodes, the block is ready for broadcast to all Peer nodes. The Orderer has the following tasks:

- Network: Utilizing gRPC [71], the Orderer accepts transactions, broadcasts, and transmits the blocks to the Peers.
- Packaging: Encapsulate the transactions into blocks based on predefined rules.
- Consensus: All Orderer nodes achieve a consensus agreement.

2.2.1 Consensus in Fabric Orderer

The consensus service in Hyperledger Fabric architecture is a pluggable module that allows for the integration of new consensus protocols, thereby enabling the operation of the Hyperledger Fabric network with these protocols. In the latest version of Hyperledger Fabric, the consensus module incorporates three implemented consensus algorithms: Solo, Kafka, and Raft. While the Raft consensus algorithm is the recommended choice, it's crucial to understand the Solo and Kafka consensus algorithms as they provide a broader context and help in comprehending the evolution of the consensus module in Fabric.

The Solo consensus algorithm operates in a network environment with a single ordering node. In this system, peer nodes send messages to the single ordering node, which then creates blocks. While Solo ensures consistency, it lacks high availability and scalability, making it unsuitable for production environments. Its inclusion in Fabric is primarily for development and testing purposes.

Kafka is a distributed streaming information processing platform designed to offer unified, high-throughput, and low-latency performance for real-time data. The previous version of Hyperledger Fabric implemented the core consensus algorithm through a Kafka cluster.

Hyperledger Fabric integrated Raft consensus algorithm in version 1.4.1. Based on the etcd library as a crash fault tolerance (CFT) ordering service, Raft follows a "leader and follower" model. Within a channel, a leader is dynamically elected from the Orderers, the `consenter set`, which is the list of the Orderer's nodes. In the

Raft protocol, follower nodes receive messages exclusively from the leader node. Raft ensures the ecosystem is able to provide services to the external world even in the presence of $(N-1)/2$ failure nodes, where `N` is the total number of Orderer nodes.

2.2.2 Fabric Consensus protocol History

The present section aims to provide a comprehensive list of Hyperledger Fabric releases and their corresponding consensus algorithms. The following versions of Hyperledger Fabric have been released:

- In 2015, IBM initialized the project called Open Blockchain (OBC) [72] [73] utilized the Practical Byzantine Fault Tolerance PBFT protocol [50].
- In 2016, The code evolved to open source and moved from the OBC repository to a Hyperledger foundation repository and named Fabric, referred to as ‘‘`v0.5-developer-preview`’’ [11] [73] using the PBFT protocol [50].
- In 2017, Versions 1.0 released of Hyperledger Fabric integrated the Kafka consensus protocol [69] as an out-of-the-box solution for production environment and Solo consensus algorithms being available for development purposes. Considering this release is the first production-ready Fabric system.
- In 2018, Version 1.4., Fabric introduced [13] the Raft consensus protocol in Etcd library [31].
- As of 2024, the latest Fabric's Version 2.5 maintains the Kafka and Raft consensus algorithms for the production environment. The Solo consensus algorithm runs on one node [13] and is used for the development environment, while Kafka is deprecated due to performance issues [74].

2.3 Orderer Node Structure

Hyperledger Fabric Orderer service, code implementation includes the server's main function and the consensus protocols. All Orderer code files are in the Fabric core in

the *orderer* folder. The entire system, including the Orderer service and the consensus protocols, is coded in the Go Programming Language [75].

The Orderer and Peer equivalent compare to a publishing and subscribe mechanism, representing the dynamic interplay between production and consumption.

The registrar's core implementation encompasses various components such as the *ChainSupport* and *Consenter*. The architecture engineering of the Consenter is pluggable. That allows for flexibility in integrating and implementing various consensus mechanisms. On the other hand, *ChainSupport* serves as a representative entity for a chain and provides the ability to reference the consensus instance associated with the same chain. Consenter creates the consensus instance that aligns with the specific consensus type. To see the complete dependency relation that the consensus instance depends on *ConsenterSupport* the flowing sections will explain each in detail.

2.3.1 Registrar

The registrar operates as a point of entry and managing for the respective channel resources. Its primary role involves managing each channel resource's access and control mechanisms. Essentially, any actions or decisions about a specific channel originate from this pivotal location (Orderer node). The GitHub source code in [76] is shown in Listing 2.1.

The Registrar struct contains the following key variable references:

1. *config* in Listing 2.1, type of *localconfig.TopLevel* This struct is the map to the *orderer.yaml* file as an external file outside the Fabric software that sets the Orderer's settings and manages the Orderer peer. The Fabric network administrator can set the configuration of the Orderer by making changes in the YAML file. Config variables provide the ability for direct access to the values in the orderer.yaml file

2. chains: The *chainSupport* variable represents and stores the chain in the Or-

derer.

3. consenters: Each consenter represents a single consensus protocol as the consenters map contains the available consensus set. Nevertheless, it is the Orderer Nodes within the Fabric network. For Fabric 2.5, the available consensus list is Solo, Kafka, and EtcdRaft consensus protocols. However, today's new Fabric 3.0 and the main branch only support Raft and SamrtBFT.

4. Ledger Factory: type of *blockledger.Factory* In order to read and create new chins in the ledger.

5. signer: It is accountable for signing the block that encapsulates the message's envelope in the Orderer node and generating the Signature Header.

6. System Channel ID and System Channel: These refer to the unique channel ID and instance of the system chain, respectively.

7. bccsp: The implementation of the blockchain cryptographic service provider Hyperledger Fabric uses.

```
47 // The Registrar acts as the individual channel resources' point of
       access and control.
48 type Registrar struct {
49   config                  localconfig.TopLevel
50   lock                    sync.RWMutex
51   chains                  map[string]*ChainSupport
52   followers               map[string]*follower.Chain
53   pendingRemoval          map[string]consensus.StaticStatusReporter
54   systemChannelID         string
55   systemChannel           *ChainSupport
56   consenters              map[string]consensus.Consenter
57     blockcutterMetrics      *blockcutter.Metrics
58   ledgerFactory           blockledger.Factory
```

```
  signer                    identity.SignerSerializer
  templator                 msgprocessor.ChannelConfigTemplator
  callbacks                 []channelconfig.BundleActor
  bccsp                     bccsp.BCCSP
  clusterDialer             *cluster.PredicateDialer
  channelParticipationMetrics *Metrics
  joinBlockFileRepo         *filerepo.Repo
}
```

Listing 2.1: Registrar struct

2.3.2 ChainSupport

Maintaining the respective channel resources by grouping interfaces capsulizes all the essential resources a channel requires, serving as a chain's representative entity. In other words, ChainSupport consolidates the resources required for a channel, serving as a representative entity for the individual chain. The GitHub source code in [77] is demonstrated in Listing 2.2. In other words, `ChainSupport` consolidates the comprehensive array of resources required for a channel, thereby serving as a representative entity for the respective chain.

```
// The resources for a specific channel are stored in ChainSupport.
type ChainSupport struct {
  *ledgerResources
  msgprocessor.Processor
  *BlockWriter
  consensus.Chain
  cutter blockcutter.Receiver
  identity.SignerSerializer
  BCCSP bccsp.BCCSP
  consensus.MetadataValidator
  consensus.StatusReporter
```

```
35 }
```

Listing 2.2: ChainSupport struct in file chainsupport.go

The ChainSupport's interfaces can be grouped as a ledger, message, consensus, and signature modules.

1. `ledgerResources`, responsible for reading the ledger info.
2. `msgprocessor.Processor`, which handles transaction processing
3. `cutter`, cutting the data into blocks to be ready for the Block Writer step.
4. `BlockWriter`, writes data blocks into the ledger.
5. `consensus.Chain`, represents the consensus instance of the Orderer as it holds the consensus protocol implementation and is responsible for starting the consensus protocol.
6. `identity.SignerSerializer`, utilized to sign the data in Orderer and create SignatureHeader.
7. `BCCSP`, it is the implementation of the blockchain cryptographic service provider Hyperledger Fabric uses.

2.3.3 Chain

Chain is an interface. Implementation is a consensus instance of a chain. The Chain functionality is to define an approach for injecting messages into the ordering process. The chain interface functions are flexible, based on the implementer's preference to architect and develop the integration of the consensus protocol, overwriting the chain interface functions. The implementer of the chain interface function is responsible for handling the ordered messages and then pushing the messages to the `blockcutter.Receiver`. to cut the blocks and write the block to the ledger. Both

functions for cutting and writing the block are available by `HandleChain`. This design enables two primary workflows:

1. Receive the messages encapsulated with envelope struct a- ordered the messages into a stream, b- cut the stream to blocks, c- the blocks are committed to the ledger. This approach is deprecated. The Orderer no longer supports as was used for solo and Kafka

2. Receive the messages encapsulated with envelope struct a- cut the messages into blocks, b- Ordered the blocks, c- Finally, commit the ordered blocks by creating and writing the block. This approach is the current etcdraft protocol uses.

The chain interface includes a set of functions the consensus team must implement in order to run the chain and receive the proposed message. The implementer of the consensus protocol's responsibility is to overwrite those functions declared in the interface. See the Chain interface functions list in Figure 2.1

```
5   type Chain interface {
6       Order(env *cb.Envelope, configSeq uint64) error
7
8       Configure(config *cb.Envelope, configSeq uint64) error
9
10      WaitReady() error
11
12      Start()
13
14      Halt()
15  }
```

Figure 2.1: Fabric Orderer Chain interface functions

2.3.4 Consenter

The Consenter interface includes one function `HandleChain`. It creates a Chain object by returning a `NewChain()` function in the `chain.go` file. Each consensus protocol has an individual consenter implementation. The Consenter interface establishes the initialization process for the ordering mechanism. The `HandleChain`

function generates and returns a reference to a Chain tailored to the specific collection of resources provided. As shown in 2.2, It is important to note that within a given execution process, `HandleChain` calls once the Orderer node starts. A visual code review for the type Consenter interface and its `HandleChain` function is demonstrated in Listing 2.3.

```
type Consenter interface {
// HandleChain is the only function required to implement the
   interface. HandleChain returns the Chain object
  HandleChain(...) (Chain, error)
}
```

Listing 2.3: Fabric Orderer Consenter interface HandleChain method

2.3.5 ConsenterSupport

`ConsenterSupport` Interface is responsible for providing the available resources to a Consenter implementation. The interface defines a set of functions as demonstrated in Listing 2.4.

```
type ConsenterSupport interface {
  identity.SignerSerializer
  msgprocessor.Processor

  // Evaluates a block's signature utilizing // an optional setting
   that can be null.
  VerifyBlockSignature([]*protoutil.SignedData, *cb.ConfigEnvelope)

  // Gives this channel's block cutting helper back.
  BlockCutter() blockcutter.Receiver

    //Return the Config Block of the current node to be shared
  SharedConfig() channelconfig.Orderer

```

```
14 // The channel configuration is obtained from the config block of
    the channel.
15 ChannelConfig() channelconfig.Channel
16
17 // Accepts a messages array,  builds the subsequent block by
    referencing the block in ledger with the highest block number.
18  // It should be noted that calling WriteBlock or
    WriteConfigBlock is required prior to calling this function.
19 CreateNextBlock(messages []*cb.Envelope) *cb.Block
20
21 // Returns null if there is no such block or a block with the
    specified number.
22 Block(number uint64) *cb.Block
23
24 // Write the block data into the ledger.
25 WriteBlock(block *cb. Block, encodedMetadataValue []byte)
26
27 // Applies the configuration modification within and Write the
    block data into the ledger.
28 WriteConfigBlock(block *cb.Block, encodedMetadataValue []byte)
29
30 //Displays an integer number of the current configuration seq.
31 Sequence() uint64
32
33 // Provides back the channel ID that corresponds to this support.
34 ChannelID() string
35
36 // Returns an integer number of the last committed block.
37 Height() uint64
38
39 // In contrast to WriteBlock, which also modifies the clock
    metadata, Append Write the block data into the ledger, but in
    raw form.
```

```
40   Append(block *cb.Block) error
41 }
```

Listing 2.4: ConsenterSupport interface

2.4 Orderer main function

The main function is the entry point of the Orderer node in the main.go file located in the Fabric core project in this directory path /orderer/common/server/main.go. Once the necessary configurations and settings have been loaded either from the YAML file or the pre-created blocks, the aim is to create the chain object and initiate the node using a particular consensus implementation. Within the Main function, several functions are called to instantiate the necessary components for the consentor and chain objects, which both support initializing the consensus protocol according to the selected consensus algorithm. The consensus module takes control of the ordering service and processing of the incoming messages. The Orderer node status is set to running once the chain and node are successfully created. This means this Orderer node is ready to participate in the network, handle transaction ordering, and create a new block according to the consensus rules defined by the chosen consensus mechanism.

The execution of the main function process involves the following steps:

1. Load the configuration file.
2. Set up the Logger.
3. Establish a local Membership Service Provider (MSP).
4. Perform core startup tasks:
    - Load the genesis block.
    - Create the ledger factory.

- Create a native gRPC server.
- Create a cluster gRPC server. If the consensus requires a cluster (e.g., raft),
- Set up the Registrar, which includes configuring the consensus plug-in, initiating each channel, and if the consensus is Raft, defining the cluster's gRPC interface processing function.
- Create a local server, which serves as the processing service for atomic broadcast. It integrates functions for broadcast processing, deliver processing, and registrar.

5. Enable profiling.
6. Start the cluster gRPC service.
7. Start the native gRPC service.

The listed functionality above all gets executed inside the main function implementation for each Orderer node under the server package.

2.4.1 Main function

The main function initializes all the required attributes for the particular Orderer node to run the consensus protocol and join the other Orderer nodes. Upon starting the Fabric network, the network administrator has the ability to designate a specific consensus protocol to govern the consensus mechanism for the Orderer. This is achieved by defining a set of key-value parameters, allowing for flexibility and customization in selecting the appropriate consensus protocol. service as demonstrated in Listing 2.5.

```
1 // The Orderer process entry point is the 'Main' function.
2 func Main() {}
```

Listing 2.5: Fabric Orderer Main method

The following steps outline the process of running the full Orderer node with the Raft consensus algorithm:

1. load()
2. initializeLogging()
3. loadLocalMSP(conf).
4. GetDefaultSigningIdentity()
5. newOperationsSystem()
6. initializeServerConfig()
7. initializeGrpcServer()
8. createLedgerFactory()
9. initializeBootstrapChannel()
10. initializeClusterClientConfig()
11. initializeMultichannelRegistrar()
12. NewServer()

1. **load()** [78] function loading the **orderer.yaml** configuration file, the initial action involves invoking the `Load()` function. This function is responsible for loading the `orderer.yaml` configuration file located outside the Fabric code as external configuration input that can be pointed within the environment variables that Fabric checks to read the file information. Upon loading the file, the configuration information is extracted and stored in the `conf` variable of type `TopLevel` struct. This process gives the Orderer node access to the configuration

information for its proper functioning. `TopLevel` directly corresponds to the Orderer config YAML. The configuration file of the `TopLevel` struct located in:

$$/fabric/orderer/common/localconfig/config.go$$

The details step helps understand the flow of the data from the out-source configuration file. The implementers and researchers of the Orderer will understand what the steps needed or changes in order to pass a customized or new variable from the `orderer.yaml` file, including the required changes in the `TopLevel` struct. The initial step in the process involves loading the `orderer.yaml` configuration file. This file contains essential configuration parameters and settings specific to the Orderer node. By loading this configuration file, the Orderer node gains access to crucial information necessary for its proper operation and behavior.

- The process of initializing the Viper component and reading the locating of the configuration YAML file by the `InitViper` function. This function sets up the Viper component, which is a popular configuration management library, and determines the path to the configuration file. By executing `InitViper`, the system initiates Viper to handle configuration-related operations and ensures that the correct configuration file is pinpointed for the following usage.
- The `ReadInConfig` operation is responsible for loading the configuration file's contents, parsing its data, and storing it within the Viper component. This process reads the configuration `orderer.yaml` file, and its values are accessible through Viper's configuration management functionality. By executing `ReadInConfig`, the system retrieves and integrates the configuration file's information, enabling subsequent retrieval and manipulation

of its settings.

- The process of parsing the Viper configuration into the `TopLevel` struct type is accomplished through the use of the `EnhancedExactUnmarshal` function. This function facilitates the transformation and mapping of the Viper data structure to the `TopLevel` structure. By executing `EnhancedExactUnmarshal`, the system ensures that the configuration data stored in Viper is accurately transformed and represented in the desired `TopLevel` format.
- In the absence of the `completeInitialization` setting, default values are utilized to supplement the missing attribute values. Additionally, upon program termination, the file path is reset to its default state to ensure the integrity of the system.

The `TopLevel` struct contains all the configurations and sub-configurations present within the `orderer.yaml` file. It serves as a full container that holds all the individual configurations, providing a direct representation of the various configuration data types and access to the entire configuration information. as demonstrated in Listing 2.6

```
// The orderer configuration YAML immediately correlates to the
   TopLevel.
type TopLevel struct {
  General              General
  FileLedger           FileLedger
  Debug                Debug
  Consensus            interface{}
  Operations           Operations
  Metrics              Metrics
  ChannelParticipation ChannelParticipation
```

```
10   Admin                    Admin}
```

Listing 2.6: TopLevel in orderer/.../config.go

2. **initializeLogging** Initializes the logging globally for the Fabric network to be in the same format and at the same logging level as it loads and sets the logging configuration at the startup of the Orderer. This function reads the environment variables. The first environment variable can set the logging level, and the second one can control and customize the logging format:

```
1 FABRIC_LOGGING_SPEC
2 FABRIC_LOGGING_FORMAT
```

   The default level for logging in the Hyperledger Fabric framework is INFO.

3. **loadLocalMSP** initializes local MSP components. The Membership service provider of the Fabric network is an essential process for a permission-type decentralized network and requires access to the crypto materials for the nodes. This function receives the config object type of `TopLevel` to load the needed properties from the `orderer.yaml` file to initialize local MSP components. First, the `GetLocalMspConfig` function must execute and load the configuration and initialize the default `BCCSP` (Blockchain Cryptographic Service Provider) for the local MSP (Membership Service Provider).

4. **newOperationsSystem** function is also called *"Orderer node running health monitor"*. During the Orderer node startup, it generates a series of metrics through the `metricsProvider`. These metrics continuously track the system's performance in real-time, such as the status of the nodes, the process by which the message envelopes status, the creation of the block status. and other critical indicators. The operating system manages and updates the data to be visible to the system administrator via a web browser using the HTTP protocol.

5. **initializeServerConfig** Load the gRPC server configuration to start the gRPC server. gRPC's configuration requires access to the `config` object mentioned earlier in this chapter responsible for loading the `orderer.yaml` file, ultimately populating the `secureOpts` to set up the TLS security authentication configuration option. This step ensures configuring the gRPC server with the necessary security measures. To enable TLS (Transport Layer Security), the `secureOpts.UseTLS` set the flag to "true". The default value is "false". However, when enabled, it indicates the necessity to read the server-side signature private key, identity certificate, and root CA certificate list. It loads the related crypto materials from the specified configuration location in the `orderer.yaml` file. The heartbeat message configuration item `KaOpts` facilitates the configuration of heartbeat parameters between the client and server.

   This item allows for the specification of heartbeat-related settings, enabling the control of intervals and other relevant parameters to ensure effective communication and synchronization between the client and server.

   The `secureOpts.RequireClientCert` flag is responsible for enabling the client certificate authentication feature. By default, the flag default value is ‘‘`false`’’. Suppose you set it to ‘‘`true`’’. In that case, the clients must connect to the server with a valid certificate for authentication, improving the security of the communication channel by guaranteeing that only authorized clients can establish a connection.

6. **initializeGrpcServer** Initializes the gPRC server instance. The *initializeGrpcServer* function is responsible for creating a gRPC server using the earlier loaded conf object encapsulates the *orderer.yaml* and *ServerConfig* parameters by the *initializeServerConfig*. It initializes and configures the gRPC server based on the specified configuration settings, ensuring the server is correctly set up and ready to handle incoming client requests. The process involves several

steps, demonstrating the necessary components and configurations to set up and initialize the gRPC server for the Orderer node communication.

- Constructing a listener on the specified IP address and port to listen for incoming connections.
- Creating an instance of `grpcServerImpl` type, referred to as `grpcServer`, using the listener and serverConfig.
- Invoking the `grpc.NewServer` to bind the server object of the gRPC server.

7. **createLedgerFactory** Create the ledger factory. The *conf.FileLedger.Location* parameter in the config object that loads the `ordererd.yaml` file specifies the location path for storing the ledger data. Trigger the execution of the `fileledger.New` function to create the ledger factory object. In the `NewProvider` function, a subdirectory with the name ‘‘`chains`’’, and inside this directory, this function creates a subdirectory with the name of the channel ID as the directory name. This directory stores the ledger data containing the block data files. The name of the block files will be according to its corresponding blockfile number. The Ledger Factory saves the block data in a structured form with separate directories for each channel's ledger data.

8. **initializeBootstrapChannel** (deprecated) initializes the system channel.

   Note: This function was deprecated and removed from the main function in Fabric Orderer 3.0. Due to the fact that there is no longer support for the system channel [79]. However, I uphold this function explanation since the long-term support (LTS) release remains in Fabric version 2.5 and still uses the system channel.

   Fabric checks the value of the `conf.General.BootstrapMethod` configuration parameter that loads from the `orderer.yaml` file to determine whether it is set to “file.” This indicates that the block data is stored using the file method.

Based on the provided configuration file or the genesis block file, the system constructs the genesis block `genesisBlock` for the system channel.

9. **initializeClusterClientConfig /clusterGRPCServer** Initialize `Cluster Client Config` when utilizing a consensus protocol that runs multiple nodes. Moreover, Fabric Orderer peers operate those consensus nodes as the etcdraft consensus protocol. Message communication is required among the consensus protocol nodes. In contrast, the Kafka and Solo modes involve solely communication between Orderer and peer nodes. Even though Kafka maintains multiple nodes, the Kafka nodes are maintained by the Kafka cluster, not embedded or maintained by the Fabric Orderer node. Consequently, under the etcdraft consensus protocol, each Orderer node operates as a **gRPC server** and a **gRPC client**. Each node runs two gRPC servers: one for handling messages submitted by peer nodes as envelopes or proposed messages and another for processing message verification and communication among other Orderer nodes. In the code, these are referred to as **grpcServer** and **clusterGRPCServer**, respectively.

10. **initializeMultichannelRegistrar** Initializes the multichannel registration. It serves the purpose of creating a manager instance, facilitating the deployment of various submodules. This manager contains Consenter instances for all consensus protocols provided in the fabric release. It runs the consenter initialization of the consensus within this function. Note that the consensus selection for the network is a configuration key-value pair in the `configtx.yaml` file. However, the return consenter object will be a function parameter for the `registrar.Initialize(consenters)`. Finally, this function returns the registrar.

    It is essential to understand what the registrar Initialize does. To narrow it down

for the consensus protocol implementer, two important functions will trigger to run:

- **func (c *Consenter) HandleChain()** The `HandleChain()` function in the *consentor.go* of the selected consensus returns *NewChain()* in the *chain.go*.
- **func (c *Chain) Start()** The `Start()` function in the *chin.go* of the selected consensus also runs to start the consensus and the chain object, indicating that the Orderer node is up and running.

```
793 func initializeMultichannelRegistrar(
794   cluster-Dialer *cluster.Predicate-Dialer,
795   srvConf comm.Server-Config,
796   srv *comm.GRPC-Server,
797   conf *local-config.TopLevel,
798   signer identity.SignerSerializer,
799   metricsProvider metrics.Provider,
800   lf blockledger.Factory,
801   bccsp bccsp.BCCSP,
802   callbacks ...channelconfig.BundleActor,
803 ) *multichannel.Registrar {
804   dpmr := &DynamicPolicyManagerRegistry{}
805
806   policyManagerCallback := func(bundle *channelconfig.Bundle) {
807     dpmr.Update(bundle)
808   }
809   callbacks = append(callbacks, policyManagerCallback)
810
811   registrar := multi-channel.New-Registrar (*conf, lf, signer,
       metrics-Provider, bccsp, cluster-Dialer, call-backs...)
812
813   consenters := map[string]consensus.Consenter{}
```

```
814
815   // the orderer can start without channels at all and have an
       initialized cluster type consenter
816   etcdraftConsenter, cluster-Metrics :=
       etcdraft.New(cluster-Dialer, conf, srvConf, srv, registrar,
       metrics-Provider, bccsp)
817   consenters["etcdraft"] = etcdraftConsenter
818
819   consenters["BFT"] = bdls.New(dpmr.Registry(), signer,
       cluster-Dialer, conf, srvConf, srv, registrar,
       metrics-Provider, cluster-Metrics, bccsp)
820
821   registrar.Initialize(consenters)
822   return registrar
823 }
```

Listing 2.7: initializeMultichannelRegistrar() in main.go

11. **NewServer** The `NewServer` function initiates a server to provide *AtomicBroadcast services*. The server consists of two handlers, one for processing the broadcast data stream and another for delivering the data stream. The server service is registered into the gRPC server `grpcServer` using the *ab.RegisterAtomicBroadcastServer* function. Fabric running the Start function of the gRPC server after *initializeGrpcServer*, starting the listening process on the specified port. Make it ready for the coming connections from gRPC clients. **Note:** that the registration of the `clusterGRPCServer` service, which is specific to multi-node consensus mechanisms such as etcdraft and BDLS, is configured in the consenter.go file for each specific consensus protocol.

```
1 //The ab.AtomicBroadcastServer that is returned by this
     NewServer function is generated by NewServer using the
```

```
    ledger reader and broadcast target.
func NewServer(
    mutualTLS bool,
    timeWindow time.Duration,
  debug *localconfig.Debug,
  metricsProvider metrics.Provider,
    r *multichannel.Registrar,
  expirationCheckDisabled bool,
) ab.AtomicBroadcastServer {
  s := &server{
    dh: deliver.NewHandler( deliverSupport{Registrar: r},
   timeWindow, mutualTLS, deliver.NewMetrics(metricsProvider),
   expirationCheckDisabled),
    bh: &broadcast.Handler{
      SupportRegistrar: broadcastSupport{Registrar: r},
      Metrics:          broadcast.NewMetrics(metricsProvider),
    },
    debug:     debug,
    Registrar: r,
  }
  return s
}
```

Listing 2.8: NewServer() in orderer/.../server.go

The NewServer function initializes and returns a new instance of the `AtomicBroadcast` server. The function is configuring the server, registering the necessary services, and starting the gRPC server to listen for incoming client connections.

2.4.2 Registrar-Consenture Wiring

In the `initializeMultichannelRegistrar` function, which we explained earlier in 2.4.1 No. 10. The function reads the consensus value that has been pre-configured

in the `configtx.yaml` file to trigger building the new consenter object for the specific consensus protocol.

```
consenters["bdls"] = bdls.New(...)
consenters["etcdraft"] = etcdraft.New(...)
consenters["solo"]= solo.New()
```

The above-selected consensus consenter initialization is the corresponding implementation of the consenter interface. by executing the `New()` function, for example $bdls.New(...)$ and $etcdraft.New(...)$ receiving the required parameters to build the consenter object. The $registrar.Initialize(consenters)$ initialize the Register, also executing the `newChainSupport` function from the `ChainSupport` by passing the initialized consenters object to the Initialize function.

$$func(c * Consenter)HandleChain()$$

The last function triggers the chain interface implementation and returns a new chain object as `NewChain()` located in the `chain.go` for the specific consensus protocol.

```
func (r *Registrar) Initialize(consenters
    map[string]consensus.Consenter) {
  r.init(consenters)
  r.lock.Lock()
  defer r.lock.Unlock()
  r.startChannels()
}
```

Listing 2.9: Initialize() in registrar.go

```
func (r *Registrar) startChannels() {
  for _, chainSupport := range r.chains {
    chainSupport.start()
  }
  for _, fChain := range r.followers {
```

```
6       fChain.Start()
7     }
8
9     if r.systemChannelID == "" {
10      logger.Infof("Without a system channel, the Registrar starts
        with %d application channels, %d consensus, and %d follower.",
11        len(r.chains)+len(r.followers), len(r.chains),
        len(r.followers))
12    }
13  }
```

Listing 2.10: startChannels() in registrar.go

2.5 pluggable consensus algorithm

Hyperledger Fabric is declared as a pluggable consensus protocol [80], which gives the ability for the end organization to integrate their preferred or customized consensus protocol as it is in charge of sending the message to the entire network peers. The pluggable architecture nature of Fabric is obtained by a predefined standard set of operations, and the implementer must implement core interfaces. The interface functions are required for the chain and consenter, start the Orderer chain, receive the submit messages, and the consensus message communication along other core interfaces must be implemented. This chapter will deeply explain the technical steps required to integrate any consensus protocol [81].

- channel artifact configuration
- func main()
- Chain Interface
- Consenter Interface

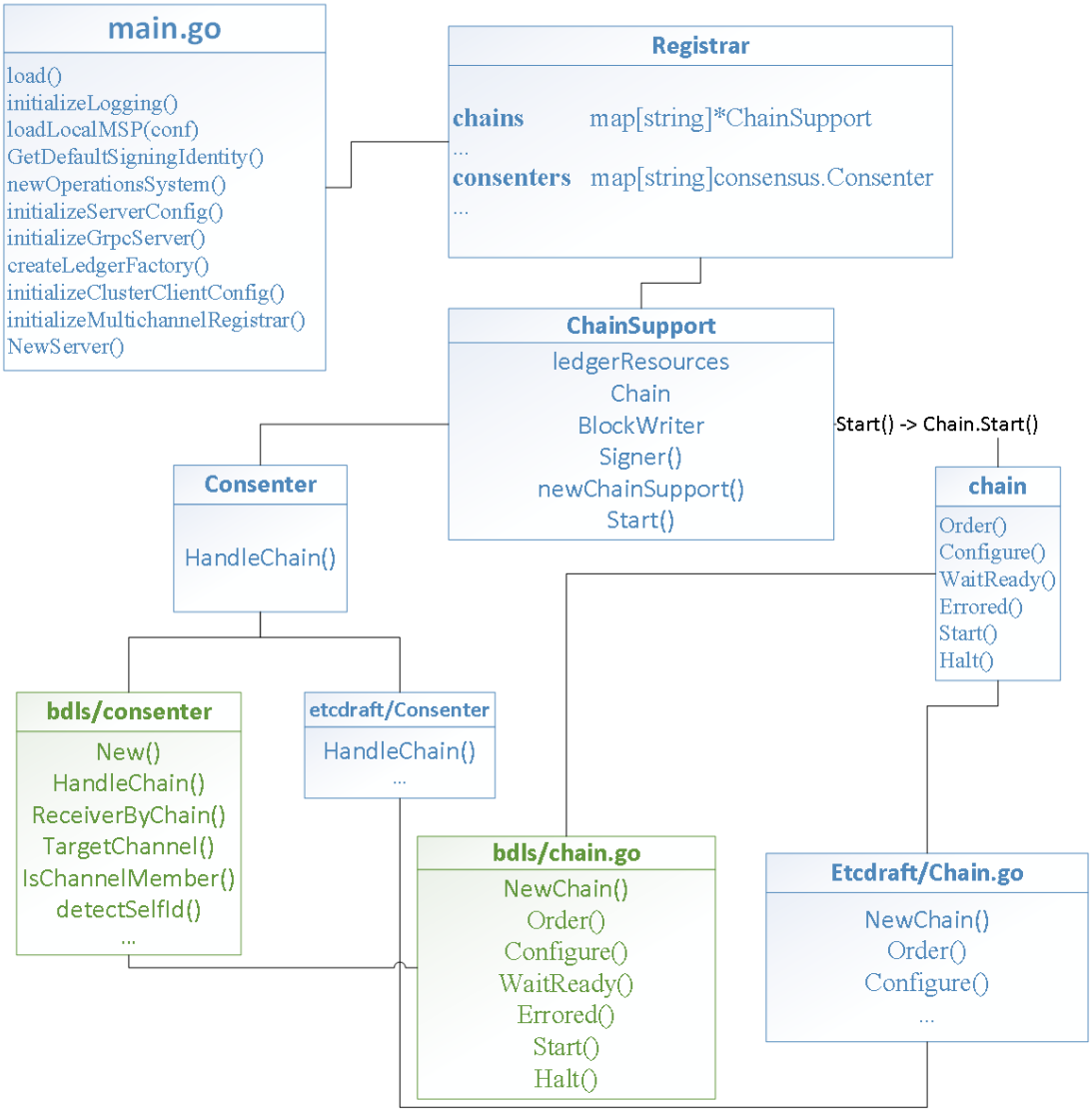


Figure 2.2: Fabric Orderer protocols Raft / BDLS. © 2024 ACM.

2.6 Fabric Raft consensus

In the Hyperledger Fabric architecture, the Orderer node is responsible for receiving transactions, packaging messages into blocks, and disseminating the newly created block to the network's peers for validation and inclusion in the blockchain. The Raft consensus algorithm is used to coordinate this process and ensure that each of the Orderer nodes comes to a consensus over the order of transactions. Raft [6] etcd [31], also known as the Raft-based etcd consensus protocol, is an implementation of the

Raft consensus algorithm used in the Hyperledger Fabric Orderer node. It is the underlying consensus mechanism for maintaining agreement and ordering transactions within the Orderer service node OSN. The etcd project, developed by the CoreOS team, provides a distributed key-value store within the Hyperledger Fabric context. Hyperledger Fabric contribution chooses etcdRaft to operate as a Fabric Orderer service consensus protocol. Fabric long-term supported release uses Raft consensus algorithms. Check figure 2.4 for the complete Raft implementation and functions relation in a data flow diagram within the Fabric Orderer node. The code core implementation for the Raft algorithm in Hyperledger Fabric resides in the Hyperledger Fabric open-source project directory:

$$fabric/orderer/consensus/etcdraft/$$

This specific location houses the essential code responsible for executing the Raft consensus algorithm within the Hyperledger Fabric framework. `package etcdraft`

2.6.1 Orderer Node Startup Process

A comprehensive understanding of the end-to-end startup process of the Fabric Orderer Node and the data flow is essential to replace or add a new consensus protocol into the Hyperledger Fabric network system. This knowledge enables developers and administrators to configure and deploy consensus protocols in the network effectively. This section provides an overview of the startup process, the data flow involved within the Fabric Orderer node, and, specifically, the consensus protocol into the Hyperledger Fabric network. Understanding the startup process helps set up the necessary configurations and ensure the consensus protocol's proper functioning.

Running the *fabric-samples/first-network* project. The `configtx.yaml` file shown in Figure 3.5.2, utilized in the creation of the `genesis.block` [82] for Orderer-related configurations.

```
Orderer: &OrdererDefaults

    # Orderer Type: The orderer implementation to start
    OrdererType: etcdraft

    Addresses:
        - orderer.example.com:7050

    EtcdRaft:
        Consenters:
        - Host: orderer.example.com
          Port: 7050
          ClientTLSCert: ../organizations/ordererOrganizations
          ServerTLSCert: ../organizations/ordererOrganizations

    BatchTimeout: 2s

    BatchSize:

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

Figure 2.3: The Orderer section in configtx.yaml file

The `genesis.block` includes the channel configuration information. Under the Orderer section, the `OrdererType` and consensus-related configuration information.

The startup command for the Orderer node is executed as *ordererstart*, which initiates the execution of the `main()` function within the main program. as described in 2.4.1. This command serves as the entry point for launching the Orderer node and initializing its core functionalities.

Reading the `genesis.block` to get the preset `OrdererType` value. Suppose the `OrdererType` is `etcdraft`. In that case, the system will utilize `clusterGRPCServer` and pass it as a parameter to `initializeMultichannelRegistrar`. In this function, the `ConsensusType` is re-evaluated. Suppose it is determined to be `etcdraft`, `initializeEtcdraftConsenter(...)` is invoked to initialize the Consenter instance object of `etcdraft`. This Consenter object is responsible for implementing the core interface called `ClusterServer`.

```
1 genesisBlock := extractBootstrapBlock(conf)
```

```
consenterType := consensusType(genesisBlock, bccsp)
switch consenterType {
    case "etcdraft":
}

```

In the `consenterType` based on the case set of the consensus, the implementation of the consenter interface implementation running the New() function located in *orderer/consensus/etcdraft/consenter.go*.

```
consenters["etcdraft"] = etcdraft.New(clusterDialer, conf, srvConf,
   srv, registrar, nil, metricsProvider, bccsp)
```

Finally, multichannel. Initialize receives all consenters as input and initializes them there. This process involves calling `HandleChain()` of `etcdraft.Consenter`, which corresponds to the core interface mentioned earlier.

The code now enters the core of etcdraft, where it automatically sets the *raftID*. The process starts with `HandleChain` and concludes with... In the `HandleChain` function, the most recent configuration data is obtained for the system channel or the channel that is currently in use from the `ConsensusMetadata`. The system sets the *raftID* based on the index of the certificate assigned to the current Orderer in this array.

The `RPC` component implements the `ClusterClient` core interface mentioned earlier, indicating a one-to-many relationship between `ClusterServer` and `ClusterClient`.

The `HandleChain` function returns the `NewChain()` function found in `etcdraft.Chain.go` to create an instance of `etcdraft.Chain`, which implements the `Chain` core interface. At this stage, all the core interfaces mentioned above in section 2.3 are now implemented.

The **newChainSupport** function is responsible for creating a chain support object `ChainSupport`, and performs the following steps:

- `ledgerResources.SharedConfig().ConsensusType()` retrieves the consensus

component type, which in this case is the “raft” type.

- `consenters[consenterType]` retrieves the consensus component object, consensus, from the consensus component dictionary.
- The `consenter.HandleChain` function is called to initialize the consensus component chain object, `consensus.Chain`, and assigns it to the `cs.Chain` field as demonstrated in Listings (2.11, 2.12).

```
func (cs *ChainSupport) start() {
  cs.Chain.Start()
}
```

Listing 2.11: start() in orderer/common/multichannel/chainsupport.go

The **chain.start** function starts the Orderer chain object that gives the ability for the Orderer node to start participating in the network and actively capable of receiving the submitted messages and handling the requests, creating the block, and writing to the ledger, also starting the consensus protocol by calling the **Start()** function of the consensus component chain within the chain support object, ChainSupport. As demonstrated in Listing 2.11.

```
func (c *Chain) Start() {
  c.logger.Infof("Starting Raft node")
    // Orderer nodes connection verification check: if there is an
   error, the Start function will not continue the initialization.

    c.configureComm();
   //Execute the above function with an 'if' condition. If an error
   occurs during the execution of the statement, check if the error
   is not null.

    // proper log for the error and continue statement

```

```
10      //This logic check indicates that the node block height> 1 once
        it joins the network.
11  //    1. The variable isJoin is set to true if the height of the
        current chain (c.support.Height()) is greater than 1, meaning
        the node is joining an existing chain.
12
13  //    2. The code then checks if both the node is joining (isJoin
        is true) and if MigrationInit (a flag in c.opts) is set to true,
        indicating that a migration is being initialized.
14
15  //    3. If both conditions are met, isJoin is reset to false,
        implying that although the node is joining, it's not considered
        a regular join due to the migration.
16
17  //    4. A log message is then generated to indicate that a
        consensus type migration is detected and a Raft the new node is
        starting on the existing channel, also displaying the current
        height of the chain.
18        // Trigger the consensus start function in the Node.go file.
19        c.Node.start(c.fresh, isJoin)
20          // Trigger running Two Go routines concurrently within the
        start function
21        go c.gc()
22        go c.run()
23        close(c.startC)
24        close(c.errorC)
25        interval := Default-Leader-less-Check-Interval
26        es := c.newEvictionSuspector()
27
28        if c.opts. Leader-Check-Interval != 0 {
29          interval = c.opts. Leader-Check-Interval
30        }
31      //initialize the PeriodicCheck struct.
```

```
32      c.periodicChecker = &PeriodicCheck{
33            CheckInterval: interval,
34            ReportCleared: es.clearSuspicion,
35            Report:        es.confirmSuspicion,
36        Condition:     c.suspectEviction,
37        Logger:        c.logger,
38        }
39      c.periodicChecker.Run()
40    }
```

Listing 2.12: start() in orderer/consensus/etcdraft/chain.go

The main three functions to start the Ordering service node are:

1. **c.Node.start()** The executing of the `c.Node.start(c.fresh, isJoin)` in line (19) in Listing 2.12 initiates the *etcdraft.Node* and launch a go routine for continuous message processing. The code was in *chain.go* then moves the raft logic to its own file `node.go` for better organization and separation of concerns. in the same package `etcdraft`.

2. **go c.gc()** referring to *chain garbage collection*, in line (21) in Listing 2.12 Invoking the `c.gc()` is responsible for creating a snapshot, recycling resources, and blocking on the `c.gcC` channel. Whenever there is a message in the `gcC` channel, the snapshot action is executed, allowing for the deletion of redundant logs or updating the snapshot.

3. **go c.run()** in Line (22) in Listing 2.12. Invoking `c.run()` initiates the chain loop and handles message processing.

2.7 Orderer messages life-cycle

The core function of the Hyperledger Fabric system is to receive messages from nodes, with the Orderer node assigned the task of distributing these messages to all

nodes across the network. Below is an outline of the key steps involved in the lifecycle of a message within the Hyperledger Fabric Orderer process.

1. The **Broadcast** function returns `Handle` function as indicated in Line (10) - Listing 2.13. It reads and processes requests from a broadcast stream accordingly, ultimately to be forwarded to the corresponding responses back to the stream. This enables both the server and the clients to communicate in both directions through the broadcast stream.

   The Broadcast processes normal messages and configuration messages separately, ensuring that each type of message is handled appropriately and in accordance with its specific requirements. This segregation allows for efficient and accurate processing of different message types within the system.

```
1  // A client sends a stream of messages to Broadcast requesting
       an order to be processed.
2  func (s *server) Broadcast(srv
       ab.AtomicBroadcast_BroadcastServer) error {
3    log.info("New Broadcast handler Starting")
4    defer func() {
5      ...}()
6    return s.bh.Handle(&broadcastMsgTracer{
7      AtomicBroadcast_BroadcastServer: srv,
8      msgTracer: msgTracer{
9      debug:    s.debug,
10     function: "Broadcast",
11     },
12   })
13 }
```

Listing 2.13: Broadcast() in orderer/common/server/server.go

2. **Order & Configure** The application sends a broadcast request message to the orderer, which forwards it to the leader node in the raft cluster. The `Order`

function handles normal messages, while the `Configure` function handles configuration messages. as demonstrated in Listing 2.14.

```
1 // The order function handles the normal message and processes
     it to the Fabric network.
2 func (c *Chain) Order(env *common.Envelope, configSeq uint64)
     error {
3   return c.Submit(env)
4 }
5 // The Configure function handles the configuration type
     message and processes it to the Fabric network.
6 func (c *Chain) Configure(env *common.Envelope, configSeq
     uint64) error {
7   return c.Submit(env)
8 }
```

Listing 2.14: Order() and Configure() functions in chain.go

3. **Submit the income message** The Submit function in Listing 2.16 encapsulates the request message into a submit struct as Listing 2.15 and assigns the request to the channel `c.submitC` as indicated in Line (6) - Listing 2.16 within the current Chain instance to process the submitted transaction message.

```
1 // submit struct used to capsulate the req
2 type submit struct {
3   req    *orderer.SubmitRequest
4   leader chan uint64 }
```

Listing 2.15: Submit struct in orderer/consensus/etcdraft/chain.go

The submit function checks whether the current Orderer node is the leader. Suppose the current node is not the leader. In that case, it must forward the message to the leader node by invoking the `forwardToLeader` function, passing the current request object that encapsulates the message. as indicated in Line

(14) - Listing 2.16.

```
1  func (c *Chain) Submit(req *orderer.SubmitRequest, sender uint64)
       error {
2  // Check if the actual node is interrupted and not running. Assign
       the result of c.isRunning() to the error variable, then check if
       it is null. If not null, update the ProposalFailures in the
       Metrics by calling Add(1). Finally, return the error.
3    leadC := make(chan uint64, 1)
4    select {
5   // Encapsulates the req in a SubmitC struct for further processing.
6    case c.submitC <- &submit {req, leadC}:
7      lead := <- leadC //Populate the lead with value from leadC
       channel
8      if lead == raft.None {
9            //update the "Proposal Failures" in the Metrics by
       calling Add(1). Return the error of absent leader within Raft.
10     }
11     if lead != c.raftID {
12 // Forward the req object to Raft leader node if current node is
       Not leader.
13       c.forwardToLeader(lead, req) {
14 //if error  != null the return from the forward To Leader function
15       }
16     }
17   case <-c.doneC:
18  // Return error message if the node has been stopped.
19     //update the ProposalFailures in the Metrics by calling Add(1).
       Return the error of stopping the chain.
20   }
21   return nil
```

Listing 2.16: Submit() in chain.go

4. **c.submitC** The `c.submitC` channel is the key channel that listens for the incoming transaction from the Submit function. The mechanism of the listener is by running the `run()` function in a go routine with an infinite loop to listen on a list of channels once receiving a value, to perform the logic on the particular channel data, `c.submitC`is one of those channels. once `c.submitC` receives a value. It performs the following actions:

   - **ordered** Passing the request message to the *c.ordered*(*s.req*) function for sorting the messages, returning the messages as batches and a boolean value in the pending variable if envelopes are pending to be ordered. as indicated in Listing 2.17 - Line (17).
   - **propose** Use `c.propose` to submit the block. as indicated in Line (37) - Listing 2.17. See Figure 2.4 for the overall process picture.

```
case s := <-submitC:
    // Once the channel submitC receives an envelope as req.
    // the case s will excute.
      if s == nil {
        continue
      }
      if soft.Raft_State == raft.State_Pre_Candidate ||
   soft.Raft_State == raft.State_Candidate {
        s.leader <- raft.None
        continue
      }
      s.leader <- soft.Lead
   // checks if the soft.Lead is  not equal to the Raft ID
      if soft.Lead != c.raftID {
        continue
      }
// This is a critical step in which all the received messages will
```

```
   be sent for ordering.
17       batches, pending, err := c.ordered(s.req)
18    // Sending the s.req.
19    // recieving: batches, which include the ordered request.
20    // pending a boolean indicator.
21    // err if exist and not equal null. logging the error and exist
      this case.
22
23
24     if !pending && len(batches) == 0 {
25        // In this check, verify if there is a pending or the
      batches do not reach the corresponding size or message number
      limit to continue receiving messages for ordering.
26              continue
27     }
28     if pending {
29     // pending is true will trigger the startTimer() function
30         startTimer()
31     } else {
32         stopTimer()
33     }
34
35 // If there are no pending batches, this means the batches have
      reached the configured limit and is ready to be proposed to the
      plug-in consensus protocol.
36
37       c.propose(propC, bc, batches...) // THIS IS A VERRY Importent
      to NOTE.
38
39 if c.configInflight {
40     log.Info("Configuration transaction received; halt transaction
      acceptance until fully committed.")
41     submitC = nil
```

```
42 } else if /* This check is with the pre deffined value*/ block In
   flight >= opts Max In flight Blocks {
43     log.Debug("halt transaction acceptance, reaches limit (%d) of
   in-flight blocks number (%d)",
44         c.blockInflight, c.opts.MaxInflightBlocks)
45     submitC = nil
46 }
```

Listing 2.17: submitC chan case in run() function, file chain.go

5. **c.ordered sorting**

   - When dealing with a configuration message, the message is cut into blocks by directly invoking the `BlockCutter.Cut()` function. This approach is adopted to handle the configuration information, divided into separate blocks.
   - During processing a normal message, the `BlockCutter.Ordered()` function is invoked to perform cache sorting. Based on the block generation rules, a decision is made whether to generate a block or not.
   - The moment the processing of c.ordered is completed, the `BlockCutter` returns the data package batches, which are the data that can be packed into blocks, along with information about whether there is any remaining data in the cache. The timer is started if there is still data in the cache that has not been generated. On the other hand, if there is no remaining data in the cache, the timer is reset. In this case, the timer handling is performed by the `timer.C` mechanism.

6. **c.propose**

   - The `propose` function packing the block invokes `createNextBlock` to package the block based on the data packets in batches.

- Then, the block is passed to the `c.ch` channel, where a separate thread is initiated to process the message. This functionality is restricted to the leader, as only the leader has the authority to propose blocks.
- In the case of configuration information, it becomes essential to annotate the status of the ongoing configuration update.

7. **c.ch** The data is transmitted to the underlying raft state machine by invoking the c.Node.Propose function as indicated in Line (10) - Listing 2.18.

   To prevent potential blocking, the leader should invoke the `Propose` method within a separate go routine as indicated in Line (1) - Listing 2.18. This allows for concurrent execution of the proposal while avoiding any potential delays or interruptions in the main execution flow.

```
1    go func(ctx context.Context, ch <-chan *common.Block) {
2  // Infinite loop that keeps listening to the channel 'ch' if a
    block or batches are received within a separate Go routine.
3      for {
4        select {
5        case b := <-ch:
6    // Marshal the block or batches to binary format.
7          data := protoutil.MarshalOrPanic(b)
8
9     // Send the Marshal data to the consensus to be proposed.
10       c.Node.Propose(ctx, data)
11       //Execute the function Node.Propose with an 'if' condition.
    If an error is raised during the execution of the statement,
    check if the error is not null and handle it accordingly.
12       // Log the error with a proper message of the blocks will
    be discarded
13
14       // Otherwise running Node.Propose with no error, a log
    statment is trigger to track the block number that been proposed.
```

```
15
16 // Exit if the context is done.
17         case <-ctx.Done():
18           // log a meesage informing that:
19         "Stopped block proposals along with the blocks ignored %d
       from queue", len(ch))
20         }
21       }
22     }(ctx, ch)
```

Listing 2.18: ch chan in run() functionfile chain.go

8. **c.Node.Propose** Proposing entails broadcasting the log to ensure widespread preservation among the nodes without immediate submission. Once the leader receives acknowledgments from more than half of the nodes, indicating successful log preservation, the leader can proceed with the submission and obtain the committed index during the next readiness phase. as indicated in Listing 2.19.

```
1 func (n *node) Propose(ctx context.Context, data []byte) error {
2 // In Propose function is the first interaction with Raft by
       proposing the Marcheled block as data for consensus.
3   return n.stepWait(ctx, pb.Message{Type: pb.MsgProp, Entries:
       []pb.Entry{{Data: data}}})
4 }
```

Listing 2.19: c.Node.Propose in:
vendor/go.etcd.io/etcd/raft/v3/node.go

9. **Save blocks**

   (a) **n.Ready()** Ready is a channel return by *etcdraft* core code. The concept of "Ready" encompasses the available entries and messages prepared for reading, saving to stable storage, committing, or transmitting to other

peers. A channel returns by "Ready" to access the current state at a specific moment. It is essential for users of the Node to invoke the "Advance" function after obtaining the state provided by "Ready." in the node.go run function `func (n *node) run (campaign bool)`.

- In the run() function, there is a (for select case) to keep checking until receiving the message in the specified channel. Once we receive the n.Ready message from the node.
- Upon receiving the "Ready" signal, it is crucial to locally store the `Entries, HardState, and Snapshot`. Raft ensures the appropriate application of these components to the state machine. As the Raft library lacks built-in storage support, it necessitates integration with the application to handle storage-related operations.
- In the presence of a snapshot, it is important to notify the `snapC` component. As indicated in Line (150) in Listing 2.20. `snapC` refers to a snapshot channel within the Raft consensus mechanism, responsible for managing the snapshotting process. Snapshotting is a critical function used to periodically capture the state of the ledger, thereby reducing the size of the Raft log by discarding older, already applied entries. This process is essential for optimizing memory usage and ensuring efficient recovery of peers in the event of failure. By utilizing snapshots, Hyperledger Fabric enhances system performance and maintains the scalability of the Raft protocol over time.
- Condition is satisfied,

$$len(rd.CommittedEntries)! = 0 || rd.SoftState! = nil$$

It signifies that there has been a change in either the `CommittedEntries`

or the `SoftState`. In such a case, updating the `applyC` channel is necessary. As indicated in Line (155) in Listing 2.20.

- Once all the processing tasks have been completed, invoking the `Advance` function will notify Raft that the current processing is finished, and it is now ready to receive the next set of updates through the "Ready" mechanism. As indicated in Line (168) in Listing 2.20.
- The `send` function initiates a remote procedure call (RPC) interface to transmit the specified "Message". It facilitates the communication and exchange of messages between different nodes in the system. As indicated in Line (170).

```
129 for {
130 select {
131 case <-raftTicker.C():
132     // To ensure that the {RecentActive} characteristics are
    not reset, capture the raft Status before ticking it.
133     status := n.Status()
134     n.Tick()
135     n.tracker.Check(&status)
136 // n.Ready() indicates that the protocol is ready to return the
    proposed data.
137 case rd := <-n.Ready():
138     startStoring := n.clock.Now()
139     // Execute the following function and check for any errors.
    If an error is returned, handle it.
140     // The storage will save Entries, HardState, and Snapshot
    from rd.
141     n.storage.Store(rd.Entries, rd.HardState, rd.Snapshot);
142    // If an error occurs, the panic function will be triggered,
    logging the error message as shown below:
143         log.Info("Data from etcd/raft failed to persist: %s",
```

```
    err)
144      duration := n.clock.Since(startStoring).Seconds()
145      n.metrics.Data-Persist-Duration .Observe(float64(duration))
146      if duration > half-Election-Timeout {
147          n.log.Warning("The synchronization process for WAL
    brought %v seconds.  Network stands scheduled to start
    elections behind %v seconds", duration, election-Timeout)
148      }
149 // if Is-Empty-Snap is not true, encapsulate the rd.Snapshot in
    n.chain.snapC.
150      if !raft.Is-Empty-Snap(rd.Snapshot) {
151          n.chain.snapC <- &rd.Snapshot
152      }
153 // applyC is channel for write  the Block, recive the block
    data the achived concensus from Raft protocol
154      if len(rd.CommittedEntries) != 0 || rd.SoftState != nil {
155          n.chain.applyC <- apply{rd.Committed-Entries,
    rd.Soft-State}
156      }
157      if campaign && rd.SoftState != nil {
158          leader := atomic.LoadUint64(&rd.Soft State.Lead)
159 // Exclusive access to this variable is necessary for etcdraft
    to function properly.
160          if leader != raft.None {
161              log.Info("Stopping the campaign,Leader already
    exist, Id: %d", leader)
162              //Next two line to stop the campaign
163              campaign = false
164              close(elected)
165          }
166      }
167 // n.Advance function call, Must be triggered within etcd-raft.
168      n.Advance()
```

```
169 // sending the messages that are extracted from the n.Ready().
170     n.send(rd.Messages)
```

Listing 2.20: n.Ready() channel in run() function in file etcdraft/node.go

(b) **n.chain.applyC**

- If the SoftState change message is received, it means that the role is about to change. Call `becomeLeader` or `becomeFollower` to change the role.
- If a message that `CommittedEntries` is not empty is received, call `c.apply` to process it.

(c) **c.apply**

- once the apply channel gets notified from the previous step the apply case in function:
  `func (c *Chain) run()` in /etcdraft/chain.go
  get triggered and run the function:
  `func (c *Chain) apply(ents []raftpb.Entry)`
- `case raftpb.EntryNormal:` Once receive a normal message, the `writeblock` function in Line (1139) / Listing 2.21, is invoked to write the block locally.

```
1126 case raftpb.EntryNormal:
1127 if len(ents[i].Data) == 0 {
1128     break
1129 }
1130 ....
1131 // Reapplying regular entries must be rigorously avoided to
     prevent writing a duplicate block.
1132 if ents[i].Index <= c.applied_Index {
1133     c.logger.Debugf("Ignored: Raft index (%d) of the
     incoming block is less than or equal to the current
```

```
    index (%d).", ents[i].Index, c.appliedIndex)
1134     break
1135 }
1136 // This next step is when Fabric unmatched the achieved
    consensus proposed data to the Fabric block.
1137 block := protoutil .UnmarshalBlockOrPanic(ents[i].Data)
1138 // Writing the block to the ledger.
1139 c.writeBlock(block, ents[i].Index) /*writeBlock*/
1140 //Set Metrics the  CommittedBlockNumber with block Header
    number))
```

Listing 2.21: Switch case for raftpb EntryNormal channel in apply function, file - chain.go

- `case raftpb.EntryConfChange:` In case the received block is a configuration block, it is written into the orderer's ledger.

  The `c.Node.ApplyConfChange(cc)` in Listing 2.22- Line (1156), calling the `ProposeConfChange` function of Raft core code is called to apply the new configuration change.

  As the Configuration Change has been introduced by a previously committed configuration block, the Orderer node unblocks the submission channel, `submitC`, to start accepting envelopes. as commented in the code block in Listing 2.22, Line (1167-1178).

```
1145 case raftpb.EntryConfChange:
1146     var cc raftpb.ConfChange
1147     // Unmarshal the configuration changes.
1148     cc.Unmarshal(ents[i].Data);
1149    //Execute the above function with an 'if' condition. If
    an error occurs during the execution of the statement,
    check if the error is not null.
1150
1151 /*** proper log for the error and continue statement ***/
```

```
1152
1153     // Keep the WAL entries on the file disk.
1154     c.Node.storage.WALSyncC <- struct{}{}
1155 // applying the configuration changes to the orderer nodes.
    Which add or remove nodes.
1156     c.confState = *c.Node.ApplyConfChange(cc)
1157     switch cc.Type {
1158     // There is no actual add node here; log a message.
1159     case raftpb.ConfChangeAddNode:
1160         c.logger.Infof("Added node %d through a
    configuration change. existing nodes in the network:
    %+v.", cc.NodeID, c.confState.Voters)
1161         // There is no actual remove node here; log a
    message.
1162     case raftpb.ConfChangeRemoveNode:
1163         c.logger.Infof("Removed node %d through a
    configuration change. Current nodes in the channel:
    %+v.", cc.NodeID, c.confState.Voters)
1164     default:
1165         log.Panic("Error- Raft configuration modification
    unsupported")
1166     }
1167 //The config block that was previously committed caused this
    ConfChange; as a result, resume submitC to receive
    envelopes.
1168     var configureComm bool
1169       if  c.confChangeInProgress.Type == cc.Type &&
1170         c.confChangeInProgress.NodeID == cc.NodeID &&
1171         c.confChangeInProgress != nil {
1172
1173         c.configInflight = false
1174         c.confChangeInProgress = nil
1175         configureComm = true
```

```
1176         // new cluster size must reported.
1177         c.Metrics.ClusterSize
1178                 .Set(length val of ConsenterIds from
    c.opts.BlockMetadata)))
1179       }
1180     shouldHalt := cc.Type == raftpb.ConfChangeRemoveNode &&
    cc.NodeID == c.raftID
1181 // To let the application maintain consuming Raft messages,
    unblock the ‘run‘ go routine.
1182     go func() {
1183         if configureComm && !shouldHalt { // If this node
    will be removed, there is no need to set up
    communication.
1184     c.configureComm();
1185             {
1186             /*Check for error if its not null to be handle
    and panic*/
1187 log.Panicf("Error: Communication configuration fail: %s",
    err)
1188             }}
1189 //Turn Off the node is the actual removal of the existing
    node by trigger c.halt().
1190 if shouldHalt {
1191     log.Info("The replica set will no longer include this
    node")
1192     c.halt()// Node is offline/deleted.
1193     return }}
1194 ()}
```

Listing 2.22: case raftpb.EntryConfChange channel in apply function, file chain.go

- In the last section of the apply function in Listing 2.24, When the accumulated received block data reaches the threshold defined by:

SnapshotIntervalSize,

$$c.accDataSize >= c.sizeLimit$$

A garbage collection (gc) signal is triggered to initiate the state machine's preparation for generating a snapshot.

```
1203   if c.accDataSize >= c.sizeLimit && ents[position].Type ==
       raftpb.EntryNormal && len(ents[position].Data) > 0 {
1204    b := protoutil.UnmarshalBlockOrPanic( ents[position].Data)
1205    select {
1206    case c.gcC <- &gc{index: c.appliedIndex, state: c.confState,
       data: ents[position].Data}:
1207      c.logger.Infof("%d bytes have been added since the last
       snapshot, surpassing the size limit of %d bytes, capturing a
       snapshot at block [%d] (index: %d), the last block number to
       be snapshotted is %d, and the current nodes are %+v",
1208    //The actual garbage collection (GC) signal is triggered.
1209 case c.gcC <- &gc{index: c.appliedIndex, state: c.confState,
       data: ents[position].Data}:
1210      log.Info("A total of %d bytes has been accumulated since
       the previous snapshot, surpassing the specified size limit of
       %d bytes. A snapshot is being captured at block [%d] (index:
       %d). The latest block number captured in the snapshot is %d.
       Current nodes include: %+v.",
1211    ...params)
1212   c.Metrics.SnapshotBlockNumber.Set(float64(b.Header.Number))
1213    c.lastSnapBlockNum = b.Header.Number
1214             c.accDataSize = 0
1215    // The default logging message with this select statement.
1216    default:
1217      c.logger.Warnf("The current state of snapshotting
       indicates that the SnapshotIntervalSize is probably too
```

```
   little")}}
```

Listing 2.23: c.accDataSize >= c.sizeLimit in apply function, file chain.go

10. **Generate a snapshot**

- The function *c.apply* dispatches the garbage collection (gc) signal to the channel *c.gcC* for further processing. Subsequently, *c.Node.takeSnapshot* and *n.storage.TakeSnapshot* are invoked in sequence to generate snapshots.

```
1219 func (c *Chain) gc() {
1220 // Infinite loop implementation in go language to select which
       channel will receive data to implement the corresponding
       select cases.
1221   for {
1222     select {
1223   // If c.gcC channel get triggered.
1224     case g := <-c.gcC:
1225       c.Node.takeSnapshot(g.index, g.state, g.data)
1226    // If c.doneC channel gets triggered, it means the node gets
       stopped.
1227     case <-c.doneC:
1228       c.logger.Infof("garbage collecting is Stopped")
1229       return}}}
```

Listing 2.24: gc() function in file: orderer/consensus/etcdraft/chain.go

- n.storage.TakeSnapshot A snapshot is generated, encompassing the term, last log subscript, and block information. The `rs.gc()` function is invoked, and if the number of snapshot files exceeds the threshold defined by `MaxSnapshotFiles`, it becomes necessary to clean up both `WAL` files and expired snapshot files.

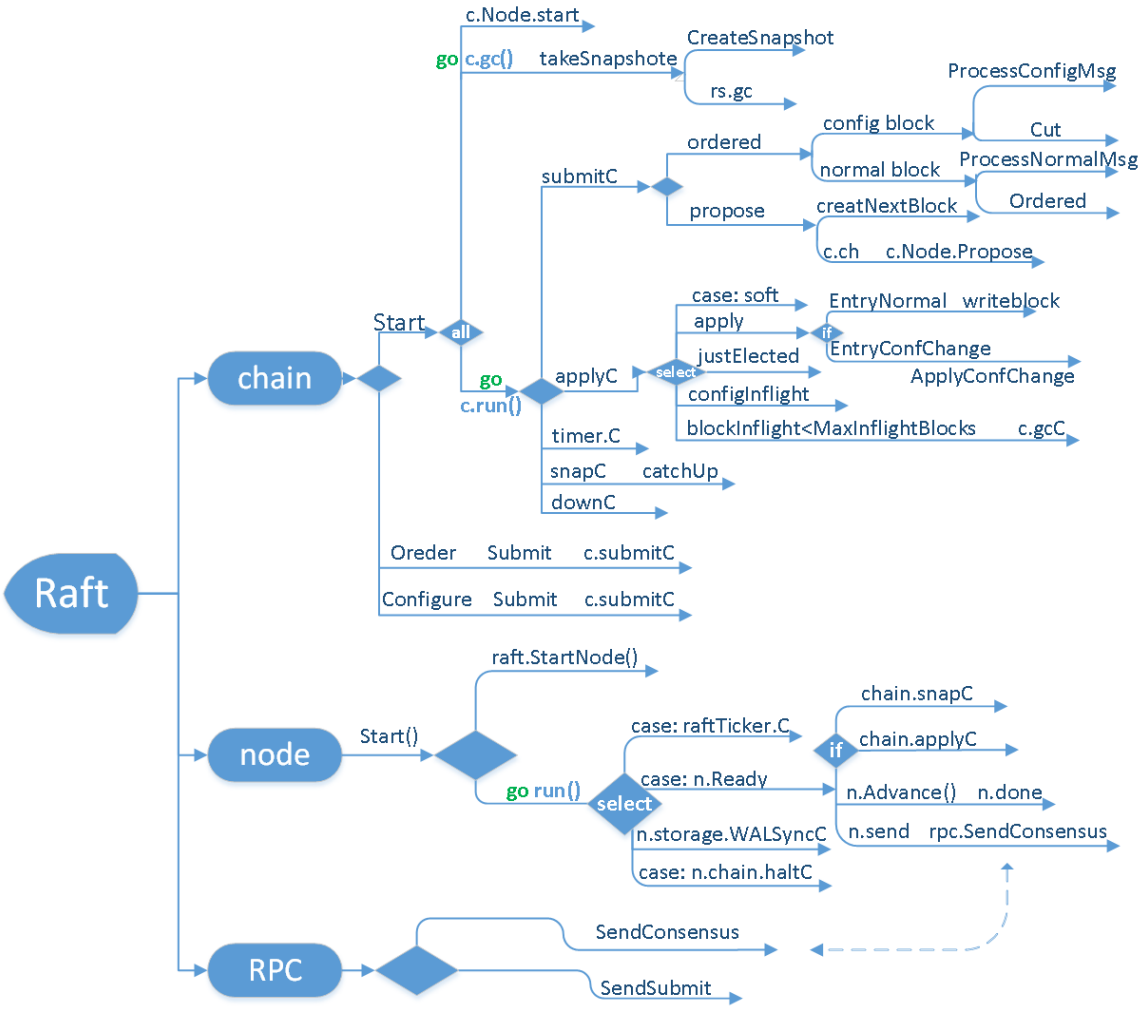


Figure 2.4: Raft flow in Fabric.
©2024 ACM. Reprinted with permission from Ahmed Al Salih, Pluggable Consensus in Hyperledger Fabric, BIOTC 2024.

CHAPTER 3: Contribution Fabric-BDLS

3.1 BDLS Introduction

The Byzantine fault-tolerant (BFT) consensus protocol is a fundamental component of blockchain systems, enabling distributed networks to agree on ordering and validating transactions without relying on centralized authorities. Despite its many advantages, the BFT protocol faces significant scalability, efficiency, and security challenges, particularly in large-scale networks with a high degree of Byzantine faults [51].

Researchers have recently proposed several improvements to the BFT protocol to address these challenges. The BDLS protocol [5] is one of the most promising BFT protocols [51]. The BDLS protocol is an enhancement to the traditional BFT consensus protocol evolved from the DLS protocol proposed by Dwork, Lynch, and Stockmeyer, for Type II networks [59] that minimizes the total number of rounds of communication needed for consensus, thereby increasing efficiency and scalability while maintaining a high degree of security. [83]

The BDLS protocol achieves its performance gains by leveraging distributed ledger signatures to reduce the number of signature verifications required during the consensus process. This technique allows the protocol to reach consensus in fewer rounds, reducing the overall time and resource requirements for achieving agreement [83].

Motivation: The integration of the Byzantine Fault Tolerance (BFT) protocol BDLS into Hyperledger Fabric Orderer presents significant opportunities for the advancement of distributed systems across various industries. Given the current limitations of the Raft protocol and the modest performance of Fabric 3.0's SmartBft, there is a clear need for a more robust and high-performance BFT solution. BDLS not only meets the crucial consensus properties of liveness and safety but also demonstrates

superior throughput (TPS) compared to existing BFT solutions. This makes BDLS an attractive alternative for industries that require high-speed, secure, and reliable distributed systems.

The successful implementation and promising experimental results of BDLS in Hyperledger Fabric Orderer highlight its potential for broader applications. Integrating BDLS into other systems or using it as a standalone protocol could significantly enhance the security and performance of distributed networks. This paves the way for further research and development, encouraging innovation in the field of consensus protocols. By leveraging BDLS, organizations can achieve greater data integrity, operational continuity, and resilience against malicious activities, ultimately leading to more secure and efficient service delivery in sectors such as healthcare, education, and finance. This motivation drives the exploration and adoption of BDLS, setting a new benchmark for the future of distributed systems and consensus protocols

3.2 BFT - Fabric Orderer

3.2.1 BFT - Fabric Related Work Review

PBFT: Hyperledger Fabric incorporates several consensus protocols. Starting in 2015, IBM initiated a project called Open Blockchain (OBC) [72] [73], which utilized the Practical Byzantine Fault Tolerance (PBFT) protocol [50]. In 2016, the code evolved to be open source. The Fabric source code moved from the OBC-IBM repository to a Hyperledger Foundation repository, named Fabric v0.5-developer-preview [11] [73], still using the PBFT protocol [50]. However, the BFT module was never released for production due to its slowness and performance issues [84] [85]. The primary challenge with PBFT and similar protocols lies in their message complexity and the substantial network traffic generated during message validation operations, which severely limits scalability. Refer to Figure 3.4 for a detailed view of the PBFT message complexity.

BFT-SMART: BFT-SMART's message pattern in typical scenarios is similar to

the PBFT protocol, lacking support for transaction pipelining. There is only one transaction in BFT-SMART can be proposed by a leader at any given moment. In the first proposal in 2018, Sousa et al. attempted to integrating the protocol to establish a Byzantine Fault Tolerant (BFT) ordering service, replacing the Kafka service with a cluster of BFT-SMART servers [12]. However, this approach was not adopted by the community due to both fundamental and technical issues, such as high message complexity and poor performance, since BFT-SMART is PBFT enhanced protocol.

Smart-BFT: In September 2023, Hyperledger Fabric announced the new Orderer version 3.0, which uses a BFT algorithm but is not an official release. Furthermore, as of 2024, the recent stable release of Fabric is version 2.5, maintains Raft consensus algorithms and no BFT model. however Hyperledger Fabric announced the Byzantine Fault Tolerant (BFT) ordering service, This protocol still suffers from the same limitations as exactly PBFT algorithm protocols as indicated using the same message pattern. Including high message complexity. Recent results show that this implementation achieves only 20% of Raft's throughput in WAN environments and 40% in LAN environments [54] [86], demonstrating significant performance drawbacks.

3.2.2 Fabric v3.0 (BFT) Limitation

Message complexity: Hyperledger Fabric recently announced a new BFT solution in its first release to incorporate Byzantine Fault Tolerance (BFT) [87]. Fabric's initial design intended to use Practical Byzantine Fault Tolerance (PBFT). However, Fabric 3.0 embedded the SmartBFT consensus library, which belongs to the PBFT family and suffers from similar message complexity issues as the network grows. In Figure in Figure 3.4 we listed a common BFT protocols are proposed for Fabric and their message complexity, including HotStuff [88] adopted in Facebook's Libra [89] blockchain, required only seven steps as we reference in our previous work [5], whereas BDLS required only four steps.

This limitation undermines the goal of maintaining a securely distributed network among institutions. For instance, healthcare systems spanning multiple hospitals at the country, state, or city level cannot effectively incorporate Fabric 3.0. Similar constraints apply to financial, retail, education, and other sectors where multiple organizations must join together to share a ledger. In BFT-SMART, the message pattern typically mirrors that of the PBFT protocol, encompassing the *PRE-PREPARE*, *PREPARE*, and *COMMIT* phases/messages. Analyzing the messages required in the network for each step: - PRE-PREPARE: One node sends the transaction to all nodes, represented as n (the number of Orderer nodes). - PREPARE: All nodes send the transaction to all nodes, resulting in n^2 messages. - COMMIT: Similarly, all nodes broadcast to all nodes, resulting in n^2 messages. Similar to MirBFT and other PBFT branches. Thus, the total messages required to achieve consensus for one transaction is $2n^2 + n$.

For a network containing 100 Orderer nodes, submitting a single transaction (i.e., creating a single block) requires:

$$\text{Total Messages} = 2(100)^2 + 100 = 20,100$$

This indicates that hundreds of orderer nodes can significantly burden the network traffic, causing delays. For the minimum network requirement of four orderer nodes in Fabric 3.0 BFT, the messages required to write a single transaction are:

$$\text{Messages} = 2(4)^2 + 4 = 36$$

Our research presents BDLS, which preserves message complexity irrespective of the growth in the number of ordering system nodes. Demonstrating with 100 Orderer nodes, BDLS maintains efficiency. In this example, P_i is the leader node, and *OSNs* are all other nodes in the ordering system. Both P_i and *OSNs* belong to the ordering

system.

$$P_i \leftarrow \text{Tx from 100 OSNs}$$

$$P_i \xrightarrow{\text{broadcast}} \text{100 OSNs}$$

$$\text{100 OSNs} \xrightarrow{\text{send}} P_i$$

$$P_i \xrightarrow{\text{broadcast}} \text{100 OSNs}$$

In each step, BDLS utilizes 100 messages sent to or broadcast by the leader to the other nodes. The total transactions for the BDLS four steps is $4n$, resulting in:

$$\text{Total BDLS Messages for 100n} = 4 \times 100 = 400$$

Comparing BDLS's 400 messages to Fabric 3.0 BFT's 20,100 messages illustrates a significant reduction in network load.

For a simple network with at least four orderer nodes, the steps required to write a single transaction using BDLS are:

$$\text{Total BDLS Messages for 4n} = 4 \times 4 = 16$$

In contrast, Fabric 3.0 requires 36 messages. Refer to Figure 3.1 for a detailed explanation of the message complexity in PBFT and its derivative protocols. This complexity arises especially during phase 2 (prepare) and phase 3 (commit), where every node communicates with all other nodes.

TPS performance limitation The throughput results of Fabric 3.0, based on the recently published paper [54] and the Fabric 3.0 announcement homepage, demonstrate significant performance limitations, in a LAN environment, SmartBFT achieves only 20% of Raft protocol performance (2,500 vs. 13,000 transactions per second) [54].

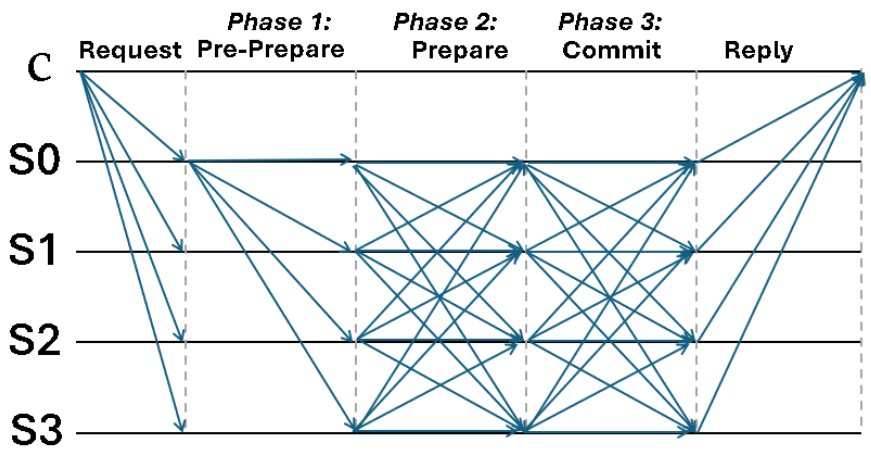


Figure 3.1: PBFT consensus algorithm - PBFT-Hyperledger Fabric full lifecycle dataflow. © 2024 IEEE.

In a WAN environment, BFT-OS achieves 40% of Raft-OS's performance (1,200 vs. 3,000 transactions per second) [54]. This is because SmartBFT [54], which is based on the Practical Byzantine Fault Tolerance (PBFT) algorithm, requires multiple rounds of communication between all pairs of nodes to ensure agreement on the order of transactions and to tolerate Byzantine faults. This communication pattern results in a quadratic increase in the number of messages as the number of nodes increases. However, not all BFT protocols suffer from these issues. Many researchers are constrained by the same protocol implementation that suffers from message complexity as listed a popular protocols proposed for Fabric in Figure 3.4. On the other hand, BDLS utilizes a digital signature scheme for cryptographic operations and message verification, avoiding these pitfalls. The BDLS TPS results demonstrated in our experimental section show a significant performance improvement. BDLS achieves 95% of Raft's throughput, illustrating that with the right implementation, BFT protocols can be both efficient and secure. This highlights the potential of BDLS as a superior alternative for achieving high performance and robust fault tolerance in Hyperledger Fabric.

Denial of Service (DoS) attack: Proven by Wang in [5] that a PBFT type

protocol cannot achieve liveness in Type II networks, which is exactly Tendermint BFT [33]. showing several attacks that can suspend and lockdown the network and reach a deadlock state and cannot achieve consensus. We pick Tendermint BFT protocol which is based of PBFT and modern popular one. presenting the message complexity and reduces the authenticator complexity to O(n) utilizing threshold cryptography in Figure 3.4.

3.2.3 BDLS Protocol

The Byzantine fault-tolerant (BFT) consensus protocol is a fundamental component of distributed systems. [47] such as blockchain systems, enabling distributed networks to agree on ordering and validating transactions without relying on centralized authorities. Despite its many advantages, the BFT protocol faces significant challenges in terms of scalability, efficiency, and security, particularly in large-scale networks with a high degree of Byzantine faults.

In partially synchronous networks, we introduce a Byzantine Fault Tolerance Protocol that robustly guarantees both safety and liveness. This approach can be extended to various applications, including State Machine Replication (SMR). The protocol is introduced as a blockchain finality gadget, featuring a unique block proposal mechanism designed to generate child blocks for finalized blocks within the BFT framework.

We're assuming there are $n = 3t + 1$ nodes labeled N_0, to, N_{n-1} in the Byzantine Fault Tolerance (BFT) protocol, with at most t of them being malicious. Additionally, each participant has a public-private key pair, with all participants are aware of the public key. We'll use the notation $\cdot i$ to indicate that participant N_i signs a message digitally.

The data flow of the proposed transaction into the BDLS node replicas requires four phases. Figure 3.3 illustrates the complete consensus process, which includes four steps: *propose*, *lock*, *commit*, and *decide*.

The BDLS protocol is established by systematically proving a sequence of lemmas,

as detailed in previous work [5]. The following points explain the consensus process protocol, which is detailed in the protocol's Figure 3.2, as depicted in Figure 3.3.

Protocol 1 BDLS (Normal Protocol Operation)

Inputs. For all $N_j \in [n]$, participant N_j propose a block B'.

Goal. All BDLS nodes (N_j, N_i) must agree on the submitted block B' from honest nodes at round r on height h.

The protocol: Request: The client submits the block B' to all BDLS nodes.

(1) **Phase 1: Leader Propose** Each participant N_j (Includes leader N_i) sends a signed message $(\langle h, r\rangle_j, \langle h, r, B'_j\rangle_j)$ to the leader N_i, where $B'_j \in \text{BLOCK}_j$ represents the highest eligible candidate block for P_j. The message $(h, r)_j$ is designated as a round-change message. Following the transmission of the round-change message, N_j ceases to accept any further messages, except for a "decide" message on a round $r' < r$.

(2) **Phase 2: Lock** P_i the leader receives messages from all consensus teams including himself. proof is a list of at least 2t+1 (nodes) same signed messages.

(a) N_i the leader receives from at least 2t + 1 participants with the same candidate block in signed messages. **Then:** All participants N_j and the leader N_i must receive this message from the leader N_i.

$$\langle \text{lock}, h, r, B', \text{proof}\rangle_i$$

A constructed digital signature on the message $\langle h, r, B'\rangle$ represents the "proof."

(b) If N_i the leader did not receive the block B ' from at least 2t + 1 participants. All candidate blocks sent to the leader N_i added in the local variable $BLOCK_i$.
The leader N_i broadcasts best-learned block B''.

$$\langle \text{select}, h, r, B'', \text{proof}\rangle_i$$

$B'' = \max\{B : B \in \text{BLOCK}_i\}$.
The "proof" contains an assembled digital signature on the message $\langle h, r\rangle$.
A constructed digital signature on the message $\langle h, r\rangle$ represents the "proof".

(3) **Phase 3: Commit** If All participants N_j (including the leader N_i) receives: $\langle \text{lock}, h, r, B', \text{proof}\rangle_i$ Then all participants N_j (including the leader N_i) send to the leader N_i.

$$\langle \text{commit}, h, r, B'\rangle_j$$

When participants receive: $\langle \text{select}, h, r, B'', \text{proof}\rangle_i$ Then participants adds $B'' \rightarrow \text{BLOCK}_j$

(4) **Phase 4: Decide** The leader N_i receives $2t + 1$ commit messages then:

(a) N_i decides on the value B'

(b) leader N_i broadcast the decided message to all participants and himself.

$$\langle \text{decide}, h, r, B', \text{proof}\rangle_i$$

Figure 3.2: BDLS Protocol (Normal Protocol Operation). © 2024 IEEE.

The BDLS protocol operates through four critical phases: leader propose, lock, commit, and decide, each designed to ensure efficient Byzantine Fault Tolerant consensus in a distributed network. These phases work in sequence to guarantee that

all non-faulty nodes reach agreement despite the presence of Byzantine actors. The following section details these phases, as each phase addresses a critical component of the consensus process. For each phase, a corresponding algorithm is presented in the appendix section, including appendix (A.1,A.2,A.3,A.4), detailing the specific steps required to execute the protocol efficiently and securely. These algorithms outline the message flows, leader roles, and conditions for progressing to the next phase, ensuring robustness and scalability in the network.

1. Request: The client is responsible for submitting the transaction to all BDLS nodes to initiate the decision-making process on the proposed block. The BDLS protocol steps commence once the client submits the block to all BFT replica members (BDLS nodes), or at least the minimum number of live replicas required by the BFT protocol, which is $3f + 1$ total nodes.

2. **Phase 1** *propose* is the first phase, all BDLS participants N_j (Includes leader N_i) send the transaction block with the proper signature to the changeable leader N_i :

   $$\langle h, r \rangle_j, \langle h, r, B'_j)_j \rangle$$

   In this context $B'_j \in \text{BLOCK}_j$ represents the highest eligible candidate block for N_j. In case of round change info. it will be within the first part as denotes $(h, r)_j$. N_j will no longer accept any additional messages, with the exception of a "decide" message pertaining to a round $r\,' < r$.

   In the context of this research, let h denote the height, r denote the round number, and B' denote the candidate block. The algorithm that represent phase one in appendix A.1.

3. **Phase 2** *lock*, the leader N_i receives total messages from at least 2t + 1 nodes, each containing the same candidate block in signed messages. The leader then

updates the block flag by adding the *lock* keyword. Finlay disseminates the block message to all Nodes. The Nodes N_j (including the leader N_i) must receive this message from the leader N_i.

$$\langle \text{lock}, h, r, B', \text{proof} \rangle_i$$

The node digitally signs and stamps the message with a *proof* flag. The algorithm that represents phase two is in appendix A.2.

4. **Phase 3** *commit* All nodes N_j (including the leader N_i) receives message: $\langle \text{lock}, h, r, B', \text{proof} \rangle_i$ Subsequently, all nodes N_j (including the leader N_i) update the block flag, adding the *commit* keyword. Then, each participant sends the updated block flag to the leader N_i with the message:

$$\langle \text{commit}, h, r, B' \rangle_j$$

The algorithm that represent phase three in appendix A.3.

5. **Phase 4** *decide* After a total node of $2t+1$ send the same message with commit flag to the leader N_i, Once the leader verify that, Then:

   (a) N_i decides on the value B' by update the flag from *commit* to *decide*.

   (b) The leader N_i broadcasts the decided message to all nodes, including himself.

$$\langle \text{decide}, h, r, B', \text{proof} \rangle_i$$

At this stage, the system is reaching consensus, indicating that, in the realm of information technology, it is deemed suitable to record the communication or restrict it to the proximate system. The algorithm representing phase four is in appendix A.4.

6. The replay step notifies the client or system of the committed block that all BDLS participants have decided on. The current BDLS protocol does not notify the client. Instead, it is the client's responsibility to call the *CurrentState()* function, which returns the last *state* (block), *block height* and *round* for checking new blocks. This functionality is successfully implemented in Hyperledger Fabric, as detailed in the *Implement Consenter Interface* section 3.4.3, specifically within the **GetLatestState** function.

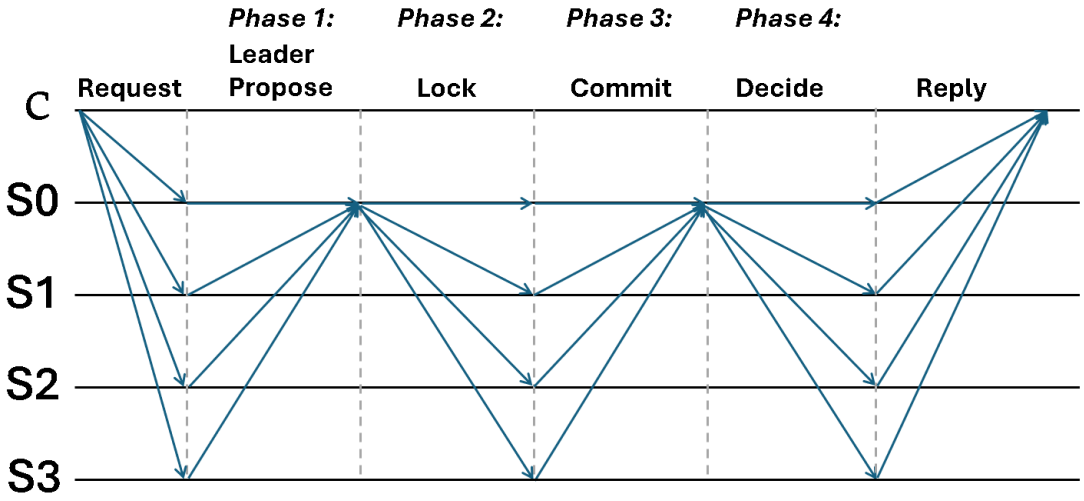


Figure 3.3: BDLS consensus algorithm - BDLS-Hyperledger Fabric full lifecycle dataflow. © 2024 IEEE.

The BDLS protocol [5] is one of the most promising consensus BFT protocols. The BDLS protocol is an enhancement to the traditional BFT consensus protocol derived from the initial DLS protocol proposed by Dwork, Lynch, and Stockmeyer, for Type II networks [59] that reduces the number of communication rounds required for consensus, thereby increasing efficiency and scalability while sustaining a high degree of security [83].

The BDLS protocol achieves its performance gains by leveraging distributed ledger signatures to reduce the number of signature verification's required during the con-

sensus process. This technique allows the protocol to reach consensus in fewer rounds, reducing the overall time and resource requirements for achieving agreement [83].

Steps	PBFT	Tendermint BFT	HotStuff BFT	BDLS	SmartBFT	MirBFT
1						
2						
3						
4						
5						
6						
7						
message complexity	$2n^2 + n$	$2n^2 + n$	7n	4n	$2n^2 + n$	$2n^2 + n$
authenticator complexity	O(n2)	O(n)	O(n)	O(n)	O(n2)	O(n2)

 : Leader broadcasts

 : All participants send messages to the leader

 : All participants broadcast

Figure 3.4: BFT protocols message complexity. © 2024 IEEE.

3.3 BDLS - Fabric integration

Technical overview of integrating BDLS that makes integrating any consensus protocol doable with this guide, along with the required code change and providing the code repository for references and practices. The change required to integrate the consensus protocol, especially BDLS, into Hyperledger Fabric applied not only to the Hyperledger Fabric core system but also to the change needed for the test network project and into the SDK project.

Hyperledger Fabric core project [90]. The change required to integrate a consensus protocol is a heavy technical effort as it contains importing the consensus protocol package and adding the particular consensus implementation.

Hyperledger fabric-samples project [91]. The change here is for running the test network, as it is not required if you plan to build the production network from scratch.

Hyperledger `fabric-sdk` project [92]. This change is required as the BDLS con-

sensus protocol's conduct works require the client to submit the transaction to all life nodes, in this case, all Orderer nodes. This change is considered a requirement for the most common BFT mechanism protocol. Contrary to the work of the existing Raft system, as the client submits the transaction throws the SDK to one Orderer node.

Fabric SDK is the library required to communicate with the Fabric node.

I will provide an overview of the reasons for a change in each project, followed by a detailed explanation of the technical changes in each project.

3.4 BDLS code change

In this section, we will delve into the significant modifications made within the Fabric project to facilitate the import and implementation of the BDLS consensus protocol. These core changes are essential for enhancing the overall functionality and performance of the blockchain framework.

We will explore the specific areas of the code that have been updated, including adjustments to the consensus layer, alterations in the communication protocols, and enhancements to transaction validation mechanisms. By detailing these changes, we aim to provide a comprehensive understanding of how the BDLS protocol integrates with the existing infrastructure and improves the system's robustness and efficiency.

Additionally, we will discuss the rationale behind these changes, emphasizing their importance in achieving greater scalability, reliability, and security within the Fabric project. This analysis will not only highlight the technical aspects of the code modifications but also demonstrate their implications for future developments in the blockchain ecosystem.

3.4.1 Import BDLS

1. Importing the Fabric project by cloning the Fabric project for the Hyperledger GitHub organization or fork the same repository to have the freedom of pushing

and tracking your changes on the cloud.

```
$ git clone https://github.com/BDLS-bft/fabric
```

2. The main function of the Orderer node in a research context requires knowledge of the selected consensus protocol that will be utilized for processing incoming messages. By incorporating the ability to select the BDLS as a consensus protocol, the Orderer node within the Hyperledger Fabric network can establish and maintain its consensus mechanism during the initialization of the network. This enhancement empowers users to choose BDLS as the preferred consensus protocol for the Hyperledger Fabric orderer. Implementing this change involves including the key-value pair 'bdls' within the variable section. See code in Listing 3.1, as $map[string]struct$ in the $clusterTypes$.

   File path: orderer\common\server\main.go *Line 70* [93]

```
var (
 clusterTypes = map[string]struct{}{
  "etcdraft": {},
  "bdls":     {},
  }
)
```

Listing 3.1: Fabric Orderer var section

   In Fabric Orderer version 3.0, the integration of BDLS into Fabric did not require any change in this section as we use the generic `BFT` value that is predefined in the cluster Types.

3. The consensus protocol algorithm software can be imported similar to any library or package into the Fabric project, executing the provided command within the Fabric's root directory at the same directory level of the `go.mod` file location. **Note:** those steps are common for importing any consensus pro-

tocol or any go package that you intended to utilize the package functions

Import the BDLS library by executing the following commands line code:

```
# Downloads and installs the BDLS Go package and its
   dependencies.
$ go get github.com/BDLS-bft/bdls
# Update the go.mod file to accurately reflect the dependencies
   utilized in the codebase.
$ go mod tidy
# Copy of the BDLS module to the vendor directory.
$ go mod vendor
```

To effectively manage dependencies in a Go project within the realm of research academics, it is crucial to employ two indispensable commands: `go mod vendor` and `go mod tidy`. The combined utilization of these commands ensures the presence of accurate dependencies and versions in your project, thereby guaranteeing its proper building and flawless execution.

By adhering to these steps, researchers and developers can seamlessly integrate the consensus package and the BDLS library into the Fabric project. This integration facilitates the utilization of their respective features and functionalities, enhancing the research endeavors.

3.4.2 Bdls-fabric package

In order to incorporate the BDLS protocol's logic and initialize the necessary consensus configuration for starting the consensus and processing of received messages, a new folder named "`bdls`" is created within the Fabric core project. This folder serves as the repository for the BDLS package and is located at the directory path:

orderer/consensus/bdls

To plug in and write the implementation of the consensus protocol logic, it is necessary to create a folder that is associated with the new protocol. This designated folder will serve as the package name decoration for the Go files within the `bdls` directory. The `bdls` folder acts as the root directory for the protocol implementation files of the BDLS consensus. All files created within this folder are considered members of the bdls package.

3.4.3 Consenter interface implementation

Create a new go file **consenter.go** in the new bdls folder as:

orderer/consensus/bdls/consenter.go

- The primary purpose of creating the `HandleChain` function as the only function that needs to be overridden in `ordere` to implement the `Consenter interface`. The function returns another function `NewChain` responsible for creating the New Chain object from the chain.go file, as explained in the next section.
- Create the **'New'** function responsible for creating the BDLS Consenter. The `New()` function is the first consensus-related function that will be called during the Orderer running. The flow from the Orderer main.go file, the `initializeMultichannelRegistrar` function passing the required parameters to the new function to create the BDLS Consenter as demonstrated in Listing 2.7 in Line (837), then pass the BDLS's consenter object to the `registrar.Initialize(consenters)` as demonstrated in Listing 2.7 in Line (850) explained in 2.4.2, 2.4.1:10 and the Orderer Node Startup Process 2.6.1.

```
consenters["bdls"] = bdls.New(clusterDialer, conf, srvConf,
   srv, registrar, nil, metricsProvider, bccsp)
```

```
2   registrar.Initialize(consenters)
3   return registrar
```

3.4.4 MessageReceiver interface implementation

MessageReceiver: This provides functions to handle the messages the Orderer node expects. In general, there are two types of messages that the OSN expects: one is the submitted message from the peer node, and the second is the messages between other Orderer nodes to verify the message. As shown in figure 3.5. In Raft, a new go file called "dispatcher.go" contains the MessageReceiver interface. For the Orderer 3.0 release, they renamed the file used in Raft to "Ingress.go". The interface's functions are the Consensus function, which handles the delivered ConsensusRequest message to the MessageReceiver. The second one is the Submit function, which handles the provided SubmitRequest message to the MessageReceiver. In the chain interface, implementing both functions is to implement the MessageReceiver interface fully.

The (OnConsensus and OnSubmit) functions notify the dispatcher of the reception messages. As shown in figure 3.6.

The related interface in the "dispatcher.go" file is the "ReceiverGetter" interface shown in figure 3.7. which includes only one function. This function is ReceiverByChain. This function returns MessageReceiver. The implementation of this function in consenter.go shown in figure 3.8, function code in Listing 3.2. Note if any missing implementations of the MessageReceiver's functions in the chain.go file, an error will occur in the ReceiverByChain function.

```
1 // For each channelID, ReceiverByChain returns the
     MessageReceiver; if not found, it returns nil.
2 func (c *Consenter) ReceiverByChain(channelID string)
     MessageReceiver {
3   chain := c.ChainManager.GetConsensusChain(channelID)
```

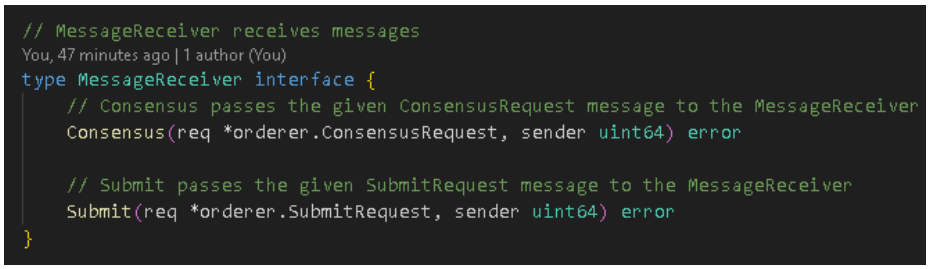


Figure 3.5: type MessageReceiver interface

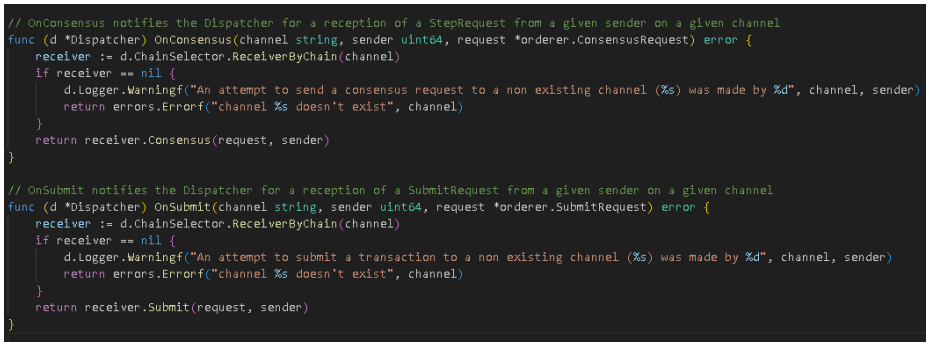


Figure 3.6: OnConsensusl() - OnSubmit() functions BDLS

```
4   if chain == nil {
5     return nil
6   }
7   if bdlsChain, isBDLS := chain.(*Chain); isBDLS {
8     return bdlsChain // error will occured if chain not
      implement the MessageReceiver interface functions
9     // In chain.go (HandleMessage & HandleRequest) for Orderer
      3.0
10    // Or Submit() & Consensus() functions.
11  }
12  c.Logger.infof("Type %v for chain %s, Not bdls.Chain, ",
      reflect.TypeOf(chain), channelID)
13  return nil
14 }
```

Listing 3.2: ReceiverByChain Function in consentor.go

The implementation of the MessageReceiver interface functions in the chain.go

```
// ReceiverGetter obtains instances of MessageReceiver given a channel ID
You, 49 minutes ago | 1 author (You)
type ReceiverGetter interface {
    // ReceiverByChain returns the MessageReceiver if it exists, or nil if it doesn't
    ReceiverByChain(channelID string) MessageReceiver
}
```

Figure 3.7: type ReceiverGetter interface

```
// ReceiverByChain returns the MessageReceiver for the given channelID or nil if not found.
func (c *Consenter) ReceiverByChain(channelID string) MessageReceiver {
    chain := c.ChainManager.GetConsensusChain(channelID)
    if chain == nil {
        return nil
    }
    if bdlsChain, isBDLS := chain.(*Chain); isBDLS {
        return bdlsChain // error if not implement the MessageReceiver interface functions
        // in chain.go (HandleMessage & HandleRequest) for Orderer 3.0
        // or Submit() & Consensus() functions.
    }
    c.Logger.Warningf("Chain %s is of type %v and not bdls.Chain", channelID, reflect.TypeOf(chain))
    return nil
}
```

Figure 3.8: ReceiverByChain Function in consentor.go

file. First, the Submit function in Listing 3.3. This function is to handle the submitted request by the peer node and pass it to the consensus protocol. This function gets the envelope encapsulated in an *orderer.SubmitRequest struct. We rap the 'req' object with a submit struct declared in the chain for simplicity of implementation.go file

```
type submit struct {
    req *orderer.SubmitRequest
}
```

Populate the c.submitC channel with the loaded request to be processed and proposed to the consensus protocol in the 'run' function. An infinite loop works as a channel listener, and if one of those channels receives a value, the logic for the code block will be triggered.

$$casec.submitC < -submitreq :$$

```
// Submit function get called by two functions, Order() and
   Config(). This function will process all submitted messages
```

```
   (normal or configuration).
func (c *Chain) Submit(req *orderer.Submit_Request, sender
   uint64) error {
select {
 // Encapsulate the req in a submit struct, then send it to the
   c.submitC channel for processing(Ordering and proposing).
case c.submitC <- &submit{req}:
    return nil
case <- c.doneC:
 //Anything populate this channel doneC, this case will update
   the Metrics and return an error.
    c.Metrics. Proposal-Failures.Add(1)
    return errors.Error("The chain Now is Down")
  }
}
```

Listing 3.3: Submit Function in chain.go

The second function must add the implementation of the MessageReceiver interface to the chain.go file. The 'Consensus' function in Listing 3.4. This function handles the message request by other Orderer nodes and passes it to the consensus protocol.

```
// Consensus implements MessageReceiver.
func (c *Chain) Consensus(req *orderer.ConsensusRequest, sender
   uint64) error {
  c.consensus.ReceiveMessage(req.Payload, time.Now())
  return nil
}
```

Listing 3.4: Consensus Function in chain.go

3.4.5 Chain interface implementation

As part of the integration process for the BDLS consensus protocol into the Fabric framework, the next step involves creating a new Go file specifically for the chain logic. This file will be named **chain.go** and placed within the newly created **bdls** package folder, located at the following directory: **orderer/consensus/bdls/chain.go.**

The purpose of the **chain.go** file is to define the core structure and functionality of the BDLS consensus algorithm as it interacts with the broader Fabric system. This file will encapsulate key elements such as the initialization of the BDLS protocol, handling of consensus messages, block proposal and validation logic, and the overall orchestration of the protocol during transaction processing.

By placing the file in the **orderer/consensus/bdls/** directory, we ensure that it fits neatly within the existing Fabric architecture, following the same modular approach used by other consensus mechanisms like Raft or Kafka. This not only maintains the consistency of the project structure but also facilitates easier maintenance and future updates to the consensus layer.

Once the **chain.go** file is established, the next steps will involve implementing the necessary methods to process blocks, manage state transitions, and ensure fault-tolerant operation in line with BDLS Byzantine fault-tolerant design. By clearly defining these processes in the **chain.go** file, we can effectively integrate BDLS into the Fabric framework, advancing towards more robust and secure consensus protocol.

To implement the **Chain** interface, we need to define all the functions specified in the interface using the **ChainStruct** type that we created earlier. This ensures that our implementation conforms to the Golang syntax standards for interfaces. Listed below are all the functions that are required to implement

the **Chain** interface.

(a) Start()

(b) Configure(config *Envelope, config-Seq uint64)

(c) Order(env *Envelope, config-Seq uint64)

(d) Halt()

(e) Errored()

(f) WaitReady()

The three main functions that operate, run, and process transactions are **Start**, **Order**, and **Configure**.

- The **Start** function serves as an instruction for the Orderer node to initiate the process of serving the chain and maintaining its up-to-date state. Moreover, make the Orderer node active to participate in the network and handle the submitted transaction by invoking the Start function. Orderer is prompted to begin executing the necessary actions to guarantee the accurate processing of transactions and the continuous synchronization of the chain with the network. Also, verify the other Orderer node's connections are configured correctly by the `configureComm` function, which can successfully accept the channel ID and a list of nodes as parameters within the `Configurator interface`. If any verification fails, an error will be returned. A BDLS config object is created for our implementation, and the required config object for the BDLS consensus is populated accordingly to initiate the protocol. These steps ensure the proper configuration of the communication layer and the initialization of the BDLS protocol. Important steps for the Orderer nodes to establish the connection and join the networks is to create a slice of cluster remote nods as indicated

in Line (10) in Listing 3.6, then configure the connection for All Orderer nodes by passing the *cluster.RemoteNode*. as indicated in Line (13) Listing 3.6. Note: Get the slice of *cluster.RemoteNode* for other Orderer nodes gets generated in a separate function by looping through the slice of the: $[] * common.Consenter$ to take a deep look at the node object created for each Orderer node check listing 3.5. generally, `RemoteNode` object contains two properties: `NodeAddress` and `NodeCerts`. The `NodeAddress` encapsulates the node ID and the node Endpoint. The Endpoint is essentially a formatted string representing the node IP Address and the Port. The second property `NodeCerts` encapsulates the server TLS certificate as `ServerTLSCert` and the client TLS certificate as `ClientTLSCert`. The BDLS-fabric running source code is available in order to review the full *remotePeers*() function source code and implementation.

```
1  nodes = append(nodes, cluster.RemoteNode{
2      NodeAddress: cluster.NodeAddress{
3          ID:       uint64(id),
4          Endpoint: fmt.Sprintf("%s:%d", consenter.Host,
       consenter.Port),
5      },
6      NodeCerts: cluster.NodeCerts{
7          ServerTLSCert: serverCertAsDER,
8          ClientTLSCert: clientCertAsDER,
9      },
10 })
```

Listing 3.5: RemoteNode Object

```
1    //Create a BDLS consensus config to validate this message
       at the correct height
2    config := &bdls.Config{
3      Epoch:          time.Now(),
```

```
4       CurrentHeight: c.lastBlock.Header.Number - 1, //0 can
        use Zero for testing.
5       StateCompare:  func(a bdls.State, b bdls.State) int {
        return bytes.Compare(a, b) },
6       StateValidate: func(bdls.State) bool { return true },
7     }
8
9   //Get the []cluster.RemoteNode for other Orderer nodes.
10    nodes, err := c.remotePeers()
11
12  // Configure the connection for All orderer nodes.
13    c.Comm.Configure(c.support.ChannelID(), nodes)
```

Listing 3.6: BDLS config cluster.RemoteNode

To begin utilizing the BDLS consensus protocol within the Hyperledger Fabric framework, the first step is to initialize the BDLS consensus object. This is done by passing the configuration parameters to the **NewConsensus** function, which is responsible for creating and configuring the BDLS instance with the necessary settings.

The **NewConsensus** function plays a crucial role in the setup process, as it ensures that the BDLS protocol is correctly instantiated based on the provided configuration object. This configuration object contains key parameters, such as the network topology, node identities, timeout settings, block generation intervals, and other protocol-specific options that are vital for the smooth functioning of the consensus mechanism.

By carefully passing the appropriate configuration object to the **NewConsensus** function, we ensure that the BDLS consensus instance is tailored to the specific needs of the blockchain network in which it operates. The code snippet below demonstrates how the **NewConsensus** function can

be used to kick-start the BDLS protocol:

```
config.Participants = append(config.Participants,
 c.participants...) // &bdls.DefaultPubKeyToIdentity())
// create BDLS consensus Object
consensus, err := bdls.NewConsensus(config)
if err != nil {
  c.logger.Error("bdls.NewConsensus", "err", err)
  return
}
// Set the BDLS consensus Latency time
consensus.SetLatency(200 * time.Millisecond)
```

In this example, the **config** object encapsulates all the necessary information required by BDLS to establish a secure and efficient consensus process. Once this function is invoked, the consensus object is fully initialized, and the protocol is ready to manage transactions, propose blocks, and achieve consensus across participating nodes.

Furthermore, this step marks the beginning of the consensus workflow, where the BDLS engine takes control of coordinating the order of transactions and ensuring that the blockchain remains consistent and resilient to faults. This initialization process sets the stage for subsequent tasks such as handling communication between nodes, managing state transitions, and ensuring Byzantine fault tolerance during consensus rounds.

By properly initiating the BDLS consensus object with the **NewConsensus** function, we lay the foundation for the successful integration of the protocol into the Hyperledger Fabric framework, enabling the blockchain network to operate efficiently and securely under the BDLS consensus model.

In the context of Hyperledger Fabric, the run function in the **chain.go**

file plays a crucial role in managing the lifecycle of the chain and handling consensus operations. To ensure that these operations are executed concurrently without blocking the main thread, we utilize Go's lightweight concurrency model by calling the run function within a *goroutine*.

```
1  go c.run()
```

In this snippet, the go keyword is used to launch the c.run() function in a new *goroutine*, where c refers to the chain instance in the Fabric ordering service. This allows the run function to operate concurrently with the main thread, enabling the system to manage consensus operations, message queues, and transaction flow without delay.

This non-blocking behavior is especially important in a distributed system like Fabric, where multiple nodes need to communicate and coordinate to achieve consensus. The chain's run function oversees the continuous operation of the chain, listening for incoming blocks, validating them, and coordinating with the consensus protocol. Running this in a separate *goroutine* ensures that other operations, such as submitting new transactionsâare not hindered, thereby improving the overall throughput of the network.

The use of *goroutines* also allows the system to scale efficiently as more transactions are processed and as the network grows. Since *goroutines* are lightweight, many can be spawned simultaneously, allowing Fabric to maintain high performance and scalability without being bogged down by heavy concurrent processes.

By leveraging Go's concurrency model in this manner, Hyperledger Fabric ensures that its chain management and consensus operations can run smoothly and efficiently, even under heavy loads. The run function operating in a *goroutine* enables seamless integration of the chain's operational tasks with the overall performance requirements of a distributed ledger.

- The **Order** function is a fundamental component in the interaction between the Orderer node and the consensus protocol, responsible for receiving and processing standard messages within the Hyperledger Fabric architecture. This function facilitates the orderly transmission of transaction data into the consensus layer, ensuring that the system can process and validate transactions in a consistent manner.

  The first parameter passed to the **Order** function is the **Envelope** object, denoted as **env**. The **Envelope** is a crucial data structure within Hyperledger Fabric, as it encapsulates the **Payload**, is the core transactional data to be ordered and committed to the ledger. In addition to the **Payload**, the **Envelope** includes a digital signature, for verifying the message's authenticity and integrity is crucial, ensuring it remains unaltered and from a legitimate source. This cryptographic guarantee is essential for securing the consensus process.

  The use of the **Envelope** structure, with its combination of **Payload** and digital signature, enables the system to securely transfer messages between nodes while preventing unauthorized or malicious modifications. By wrapping the transaction data in an **Envelope**, the consensus protocol can confidently process and validate messages, ensuring that only authenticated and verified transactions are accepted for ordering and inclusion in the blockchain.

  The role of the *Order* function in this process is to act as the entry point for these encapsulated messages, triggering the subsequent steps of validation and ordering within the consensus mechanism. This makes the Order function not only a critical point of interaction between the Orderer node and the consensus protocol but also a key component in ensuring the integrity and reliability of the overall transaction flow within the Hyper-

ledger Fabric network, as illustrated in Listing 3.7.

```
// The envelope holds the message submitted by the client.
type Envelope struct {
  // The Payload received in marshaled format.
  Payload []byte `protobuf:..."`
  // An author signature in the Payload header signature.
    Signature            []byte   `protobuf:"..,omitempty"`
  XXX_NoUnkeyedLiteral struct{} `json:"-"`
  XXX_unrecognized     []byte   `json:"-"`
  XXX_sizecache        int32    `json:"-"`
}
```

Listing 3.7: Envelope struct

- The **Configure** function is responsible for processing messages related to network configuration updates, typically initiated by an administrator. These updates include operations such as adding or removing nodes from the blockchain network. The **Configure** function ensures the system remains consistent and functional while applying these changes. It plays a crucial role in maintaining network flexibility and scalability.

  This function must adhere to the Chain interface, located in *consensus.go*, which defines the necessary methods for consensus protocols within Hyperledger Fabric. The Chain interface provides the structure for block proposals, message handling, and configuration management, ensuring seamless interaction between the consensus protocol and the ordering service.

  The implementation of the **Configure** function for the BDLS consensus protocol requires creating a chain.go file in the following directory:

```
orderer\consensus\bdls\chain.go
```

By implementing the **Chain interface** in *chain.go*, the BDLS protocol can efficiently manage network configuration changes while ensuring compatibility with the Fabric ordering service. This approach allows the BDLS consensus to handle dynamic network updates, ensuring smooth operation while adding or removing nodes and providing resilience in the face of evolving network conditions.

Below is a core function required to initialize the Orderer node, specifically the **NewChain** function. Additionally, other essential functions, such as **GetLatestState**, must also be utilized within the BDLS protocol described in detail below.

- The **NewChain** function is designed to instantiate a new chain object. This function is called upon as the return value from the "HandleChain" function. It is employed to manage and initialize chain objects, ensuring seamless integration and functionality in blockchain operations. By utilizing this mechanism, the **NewChain** function facilitates the creation and organization of chain structures, contributing to the system's overall efficiency and operability within distributed ledger environments.
- **GetLatestState** function Within the *chain.go* module, verifying the BDLS state value is necessary to determine the appropriate time to write a block. The algorithm must achieve a consensus and update the state value to be **decide**.

  To access the latest BDLS state, you can utilize the *CurrentState()* function in BDLS core code. Additionally, A new function called *GetLatestState* returns three parameter values: height number, current round, and the latest state. The *CurrentState()* shown in Listing 3.8. Also, check the architecture diagram of BDLS flow in Fabric in Figure 3.9.

```
func (c *Consensus) CurrentState() (height uint64, round
    uint64, data State) {
  return c.latestHeight, c.latestRound, c.latestState}
```

Listing 3.8: CurrentState returns the latest (height, round, state)

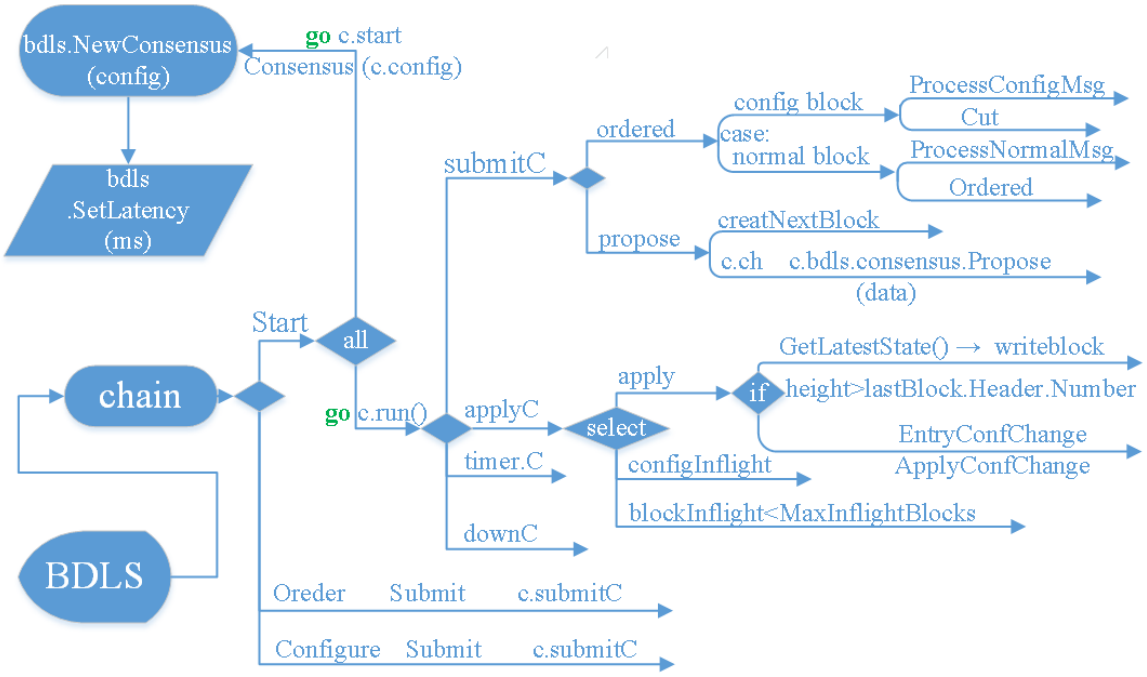


Figure 3.9: BDLS flow in Fabric

- Adding required interfaces:

  RPC interface: The RPC interface encapsulates the functions that are utilized by the Fabric Orderer node for sending the message. For fabric OSNs, there are two types of message transport. The first one is by sending the message to the leader Orderer node; however, in such a BFT-type consensus protocol, the BDLS protocol library will handle forwarding the submitted request to the leader internally, so there is no need for the Orderer node to forward the request to the leader node. The second type of message transport is the consensus message among the protocol peers to verify the message and agree on the same message to achieve consensus.

BlockPuller interface: The `BlockPuller` component is utilized to retrieve or fetch blocks from other `OSNs` (Ordering Service Nodes). This functionality allows the system to efficiently acquire blocks from remote OSNs, enabling the synchronization and replication of block data across the network. By employing the `BlockPuller`, the system can retrieve blocks from various sources, ensuring the availability and consistency of data throughout the distributed network nodes.

Configurator interface: The `Configurator` module is responsible for configuring the communication layer at the initiation of the chain. It ensures that the necessary settings are appropriately set up to facilitate seamless communication between the cluster of remote nodes within the chain. The `Configurator` is crucial in establishing the communication infrastructure and ensuring that the chain's participants can effectively exchange information and interact. In Fabric Orderer 3.0, we successfully implemented the connection among the network without the need to create or implement this interface as we handle the node connection configuration in the Start function.

Note: In Fabric Orderer version 3.0, the following changes are not required: the BDLS-Fabric integration uses the default BFT value, and the changes are required only in the *Main.go* function and create the required implementation files in:

$$orderer/consensus/bdls/$$

4. This is the fourth step required for integrating a protocol into Hyperledger Fabric: adding the protocol name to the configuration file, **config.go** adding the type key for BDLS-based consensus in the constant declaration $Bdls = \text{“}bdls\text{”}$ creating profile key value, [94].

Note: In Hyperledger Fabric Orderer 3.0, we used the BFT key value that comes out of the box and ignored the entire step four and five implementations.

(a) The profile `ChannelUsingBdls` is designed for testing the BDLS-based ordering service using the channel participation API. It serves as a reference for configuring and assessing the service's capabilities within application channels and development environments.

Note: In this research experiment, the BFT key profile provided within Fabric Orderer version 3.0 is used as a default out-of-the-box solution for BFT protocols.

```
// SampleDevModeBdlsProfile
SampleDevModeBdlsProfile = "SampleDevModeBdls"

// SampleAppChannelBdlsProfile ordering service sample profile.
SampleAppChannelBdlsProfile = "SampleAppChannelBdls"
```

Within the Orderer struct, it is important to include the BDLS key and a pointer to the BDLS's `ConfigMetadata`. The `bdls.ConfigMetadata` is imported from an independent project that has been incorporated into the Fabric codebase. This metadata encompasses the generated Protocol Buffers (protobuf) files essential for implementing the BDLS protocol. Including the BDLS key and the `ConfigMetadata` pointer ensures seamless integration of the BDLS protocol and grants access to the relevant protobuf files required for the proper functioning of the Orderer component.

```
// Channel-related configuration is contained in the orderer.
type Orderer struct {
OrdererType       string                    `yaml:"OrdererType"`
Organizations     []*Organization           `yaml:"Organizations"`
Addresses         []string                  `yaml:"Addresses"`
```

```
Capabilities      map[string]bool           `yaml:"Capabilities"`
Bdls              *bdls.ConfigMetadata      `yaml:"Bdls"`
BatchSize         BatchSize                 `yaml:"BatchSize"`
MaxChannels       uint64                    `yaml:"MaxChannels"
BatchTimeout      time.Duration             `yaml:"BatchTimeout"`
ConsenterMapping  []*Consenter           `yaml:"ConsenterMapping"`
EtcdRaft          *etcdraft.ConfigMetadata `yaml:"EtcdRaft"`
Policies          map[string]*Policy        `yaml:"Policies"`
}
```

Including the case of BDLS in the `completeInitialization` function in [94], demonstrated in the *config.go* file path in the Fabric project:

internal/configtxgen/genesisconfig/config.go

5. In **util.go** file, appending the function `MarshalBDLSMetadata` [94] is responsible for serializing BDLS metadata, as demonstrated below. It expects the user to specify the configuration significance for client and server certificates, pointing to the local path where these files are permanently kept on the file system. Subsequently, the function loads these certificate files into memory for further processing. This approach ensures that the BDLS metadata is serialized correctly and prepared for subsequent operations. The *util.go* file location in the Fabric project:

common/channelconfig/util.go

```
// MarshalBDLSMetadata serializes BDLS metadata.
func MarshalBdlsMetadata(md *bdls.ConfigMetadata) ([]byte,
   error) {
  copyMd := proto.Clone(md).(*bdls.ConfigMetadata)
  for _, c := range copyMd.Consenters {
    // After loading these files into memory, anticipate that
```

```
    the user will set the client/server cert configuration value
    to the path where they are locally stored.
6     clientCert, err := os.ReadFile(string(c.GetClientTlsCert()))
7     Check err if not null {
8       return null with the error message("Client cert cannot
    loaded for the consenter host:port %s:%d: %s", c.GetHost(),
    c.GetPort(), err)
9     }
10    c.ClientTlsCert = clientCert
11 // loads the server certificate by reading the pre defined file
12    serverCert, err := os.ReadFile(string(c.GetServerTlsCert()))
13    Check err if not null {
14      return null with the error message ("Server cert cannot
    loaded for the consenter host:port %s:%d: %s", c.GetHost(),
    c.GetPort(), err)
15    }
16    c.ServerTlsCert = serverCert
17  }
18  return proto.Marshal(copyMd)
19 }
```

3.5 Fabric test Network

Hyperledger Fabric test network is a pre-configured project using a minimal set of bash scripts. It offers a platform for testing smart contracts and applications in various programming languages. Provide and run the network on the selected ordering service provider. Moreover, it is capable of running a multi-node based on the user configuration. The fabric-samples repository project includes out-of-the-box test network projects. We will demonstrate in this section the ‘‘`test-network-nano-bash`’’ project. The nano-bash project allows developers and researchers to bring up the network using the Fabric source code directly with no need to build Docker images. The functionality of the running Fabric network is equivalent to the docker-based Test

Network offering multi-node, and the TLS layer is enabled. It required a one-time configuration for the development environment. It advocates for Fabric's developers and researchers to make changes in the Fabric source code and use the generated Fabric binaries. That will eliminate the time spent building the fabric images. Equally important is using the Fabric source code bindery that could generate quickly rather than the image binary [95].

3.5.1 Cryptogen Tool

Cryptogen is a utility tool for generating crypto material for Hyperledger Fabric network nodes in the developer environment to generate self-signed keys. The `cryptogen` utility tool is a binary tool that provides the ability to generate a sample configuration file.

```
cryptogen showtemplate > crypto-config.yaml
```

The new `crypto-config.yaml` or any name based on your choice makes it easy for review and modification as the output in YAML format. to adjust the number of the Orderer and network nodes. In order to generate the crypto materials, you specify two parameters: First, the input configuration YAML file, and Second, the output directory for the actual crypto materials.

```
cryptogen generate  --config=./crypto-config.yaml
   --output=./crypto-config.
```

The `cryptogen` first checks for the environment variable in the operating system. The environment variable $FABRIC_CFG_PATH$ specifies the file path to the YAML configuration file, whether located on the system or within the network, to facilitate the loading of the configuration. If this environment variable is not set, the `cryptogen` tool will check for the configuration YAML file in the directory from which it is being executed. It also provides the ability to extend the existing generated materials without regenerating or touching the old crypto. For our work, we have recently used The `test-network-nano-bash` project. As mentioned earlier, it provides a set of

out-of-the-box script for the network setup. The `generate_artifacts.sh script` is used to generate the entire certificates for the Fabric network nodes. The nature of the BDLS protocol requires a specific setup. Regarding that, We pass the BFT key while running the `generate_artifacts.sh script`.

```
./generate_artifacts.sh BFT
```

Incorporating the BFT key with the aforementioned script involves the establishment of an environment variable to facilitate the loading of the configuration through a designated `configtx.yaml` file. This will trigger the export of the environment variable $FABRIC\_CFG\_PATH$, assigning it the path of the current BFT directory, which contains the corresponding `configtx.yaml` file required for the BFT configuration.

3.5.2 Configtxgen Tool

The `configtxgen` command is a binary utility tool that enables Fabric developers to create and examine artifacts associated with channel configuration [79]. These artifacts play a crucial role in defining the structure and properties of a channel within the Hyperledger Fabric network [96]. The specific contents of the generated artifacts are determined by the configuration specified in the *configtx.yaml* file.

By configuring the various parameters and settings in the *configtx.yaml* file, the system administrator can tailor the channel's characteristics, including the participating nodes for each organization, their governance policies, the consensus algorithm governing the ordering service, and other critical configuration details. The `configtxgen` command, in conjunction with the `configtx.yaml` file, empowers users to generate and examine artifacts associated with channel configuration, providing a toolset for channel configuration.

Note: Fabric 3.0 BDLS integration and testing use the "test-network-nano-bash" project for testing and configuring the Fabric network artifacts. The bellow section to the end of this chapter is for a deep understanding of the use of the `configtx` tool for manual network artifact setup, which is what we used for the BDLS-Fabric integration

progress prior to fabric 3.0 and prior to the "test-network-nano-bash" project.

In the Hyperledger Fabric ecosystem, there is flexibility in overriding the generated configuration by setting environment variables for any primary section in the *configtx.yaml* file. When an environment variable is set, it takes precedence over the corresponding configuration in the YAML file. For example, the network can be forced to use the BDLS protocol or the generic BFT by setting the environment variable:

$$CONFIGTX\_ORDERER\_ORDERERTYPE = bdls$$

, thereby overriding the default ordering service configuration.

configtx.yaml The configuration settings for the BDLS `ordererertype` is defined within the `Orderer` section. This configuration is necessary when opting for the BDLS-based implementation in the network. In the `Orderer:Bdls:Consenters` section, a list of BDLS replicas for the network is specified. It is important to note that for the BDLS-based implementation, each replica is expected to function as both an Orderer service node `OSN` and a replica. Consequently, the object displayed for host:port within this replica set is appropriate and must be duplicated beneath the Orderer.Addresses key, illustrated in Figure 3.10. These configuration specifications ensure the correct deployment and functionality of the BDLS-based Orderer. In the `configtx.yaml` file, an `Options` section is included to provide developers with the flexibility to set a value that can be utilized as a differentiation factor within the *Option* **struct** in the chain.go implementation. In the Go (Golang) programming language, a **struct** is a composite data type that facilitates grouping related values of different types. This option allows developers to configure specific attributes or behaviors based on the provided value. By incorporating this option in the *configtx.yaml* file as illustrated in Figure 3.10 last line, researchers and developers can easily customize and adjust the behavior of the chain implementation based on their specific requirements and experimental needs. This enhances the configurability and adapt-

ability of the system, enabling researchers to explore various scenarios and evaluate the distributed network performance. The *configtx.yaml* in the fabric project can be found at:

$$sampleconfig/configtx.yaml$$

The `fabric-samples` project repository [97], which utilizes the Fabric test network

```
################################################################################
#
#   ORDERER
#
#   This section defines the values to encode into a config transaction or
#   genesis block for orderer related parameters.
#
################################################################################
Orderer: &OrdererDefaults

    # Bdls defines configuration which must be set when the "Bdls"
    # orderertype is chosen.
    Bdls:
        # The set of Bdls replicas for this network. For the Bdls-based
        # implementation, we expect every replica to also be an OSN. Therefore,
        # a subset of the host:port items enumerated in this list should be
        # replicated under the Orderer.Addresses key above.
        Consenters:
            - Host: bdls0.example.com
              Port: 7050
              ClientTLSCert: path/to/ClientTLSCert0
              ServerTLSCert: path/to/ServerTLSCert0
            - Host: bdls1.example.com
              Port: 7050
              ClientTLSCert: path/to/ClientTLSCert1
              ServerTLSCert: path/to/ServerTLSCert1
            - Host: bdls2.example.com
              Port: 7050
              ClientTLSCert: path/to/ClientTLSCert2
              ServerTLSCert: path/to/ServerTLSCert2
            - Host: bdls3.example.com
              Port: 7050
              ClientTLSCert: path/to/ClientTLSCert3
              ServerTLSCert: path/to/ServerTLSCert3

        # Options to be specified for all the Bdls nodes. The values here
        # are the defaults for all new channels and can be modified on a
        # per-channel basis via configuration updates.
        Options:

            CurrentHeight: 0
```

Figure 3.10: configtx.yaml - BDLS configuration settings

[95], has undergone a configuration folder restructuring. The configuration folder is now located in the */test-network/configtx* directory instead of the repository's root directory. This updated configuration folder now includes only the *configtx.yaml* file,

which contains essential configuration parameters for the network.

Furthermore, the process of setting the environment variable $FABRIC_CFG_PATH$ has been automated to eliminate the need for manual interaction. Upon executing the *network.sh* file to start the test network, the environment variable will be automatically set, ensuring the correct configuration path is utilized.

```
FABRIC_CFG_PATH=${Fabric root path}/configtx
```

Suppose the environment variable $FABRIC_CFG_PATH$ has not been set up. In that case, the *configtxgen* binary command will attempt to locate the *configtx.yaml* file in the system directory from which the command is executed. To set the utilization of a new consensus algorithm in the network, a specific change needs to be made in the *configtx.yaml* file. This change involves modifying the Orderer section by setting the *OrdererType* to "Bdls" and adding a BDLS key section under the Orderer. Within the BDLS section, a *Consenters* subsection should be included, which comprises the details of the participating nodes. Each Consenter entry should include the *Host, Port, ClientTLSCert* and *ServerTLSCert* parameters, specifying the corresponding network configuration for the BDLS consensus algorithm.

This configuration change allows for the explicit declaration of the BDLS consensus algorithm as the chosen consensus mechanism in the network. By specifying the *Consenters* and their associated parameters, the network can effectively establish the necessary communication and security configurations required for the BDLS consensus protocol. This modification to the `configtx.yaml` file plays a crucial role, particularly in enabling the adoption of the BDLS consensus algorithm.

- Channel Transaction file

  There are two profiles for BDLS created in the `configtx.yaml` file, as a subsection under the Profiles main section as illustrated in Figure 3.11. To build the Tx file using the `SampleAppChannelBdls` profile.

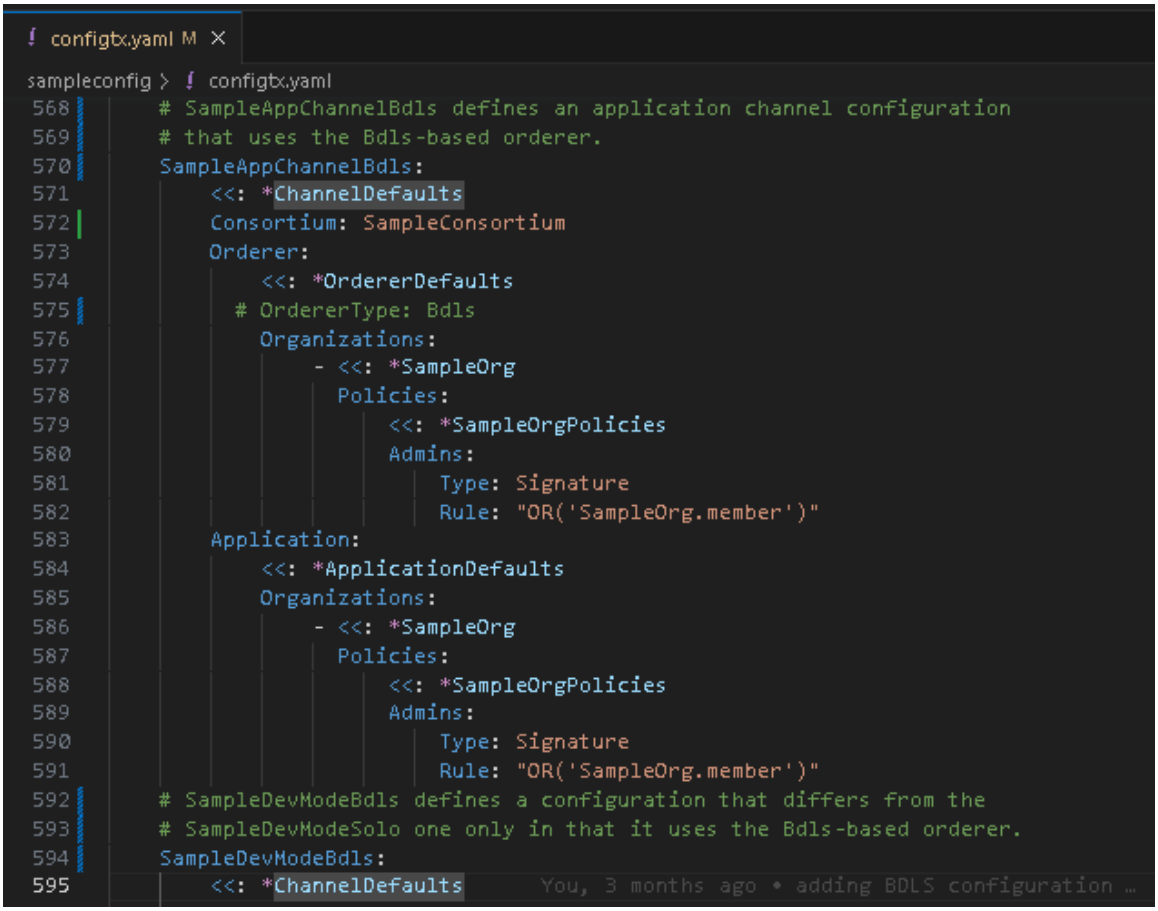


Figure 3.11: configtx.yaml - BDLS profile's configuration

```
configtxgen
    -outputCreateChannelTx   ./channel.tx
    -profile                 SampleAppChannelBdls
    -channelID               myChannel
```

Listing 3.9: Generate Channel Transaction file

To examine the TX Transaction file generated in binary format and convert it into a human-readable JSON file, the following command should be executed.

```
configtxgen
    -inspectCreateChannelTx channel.tx > ./temp/channel.json
```

Listing 3.10: Inspect Channel Transaction file

- Generate the genesis block

```
configtxgen -outputBlock ./first-channel.block
            -profile TwoOrgsChannel
            -channelID myChannel
```

Listing 3.11: Generate the genesis block

To examine the genesis block file generated in binary format and convert it into a human-readable JSON file, the following command should be executed.

```
configtxgen -inspectBlock first-channel.block >
   ./temp/channel.json
```

Listing 3.12: inspect the genesis block

BDLS The type key for BDLS-based consensus. that will be used in *TopLevel* consists of the Golang *struct*, used by the configtxgen tool. Include BDLS for the Orderer struct as $Bdls * bdls.ConfigMetadata`yaml:"Bdls"`$

```
internal/configtxgen/genesisconfig/config.go
```

Build Fabric Docker Images and Binaries

```
# Strop all running docker container
docker stop $(docker ps -a -q)
docker system prune -f ; docker volume prune -f ;docker rm -f -v
   $(docker ps -q -a)
docker rmi -f $(docker images -q)
make clean
make unit-test
make configtxgen configtxlator cryptogen orderer peer osnadmin
   docker
```

Starting from Fabric version 2.3, the command `osnadmin` introduces new functionality. Administrators can utilize the *"osnadmin channel"* command line binary tool to execute operations pertaining to channels on an orderer. The *osnadmin* provides several operations on the Orderer node to control Orderer membership, such as giving

the ability to (add or remove) the Orderer from the channel. Also, it can display the list of the channels in which an Orderer has participated [98]. In order to set up the Admin endpoint for every orderer and activate the channel participation API, adjustments must be made in the `orderer.yaml` file [98].

Two distinct categories of build artifacts are generated during the build process: binaries and Docker images. These artifacts are located within the newly created build folder, which is situated in the root directory of the fabric project. The binaries can be found in the `build/bin/` directory, while the Docker images reference are stored in the `build/images/` directory.

To verify the integrity of the build artifacts for the binary, proceed to the build/bin folder and examine the list of binary artifacts. Refer to Figure 3.12 for the comprehensive list of expected binary artifacts. To validate the generated Docker images, execute the command "docker images" and ensure that the generated Docker images are displayed, aligning with the visual representation depicted in Figure 3.12.

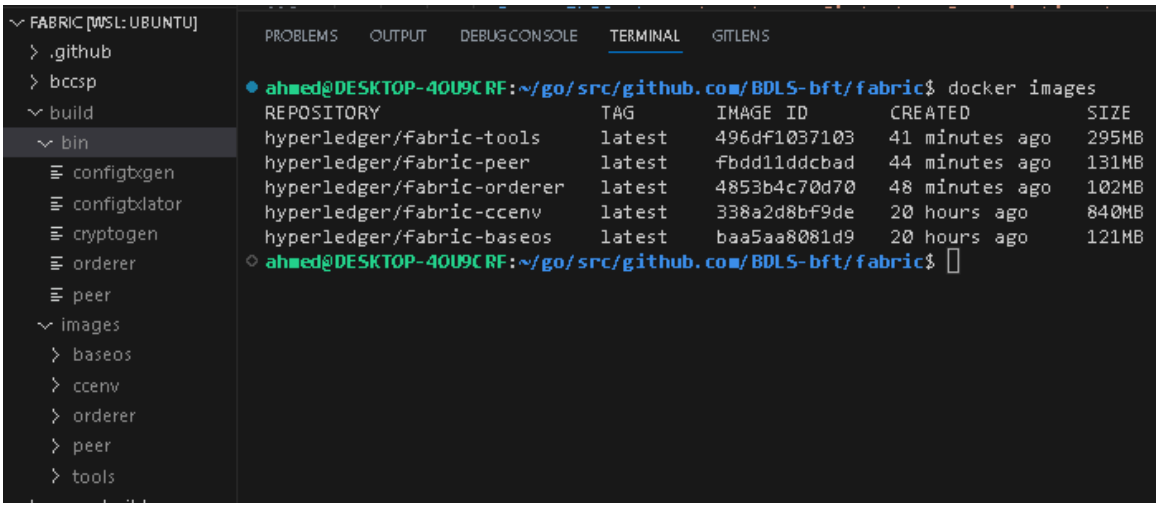


Figure 3.12: Build Fabric binary and docker images

For the independent construction of the Orderer docker image, you can execute the following command-line instruction. This process allows for the creation of the Orderer docker image in isolation.

```
cd images/orderer/
docker build -t hyperledger/fabric-orderer:latest .
```

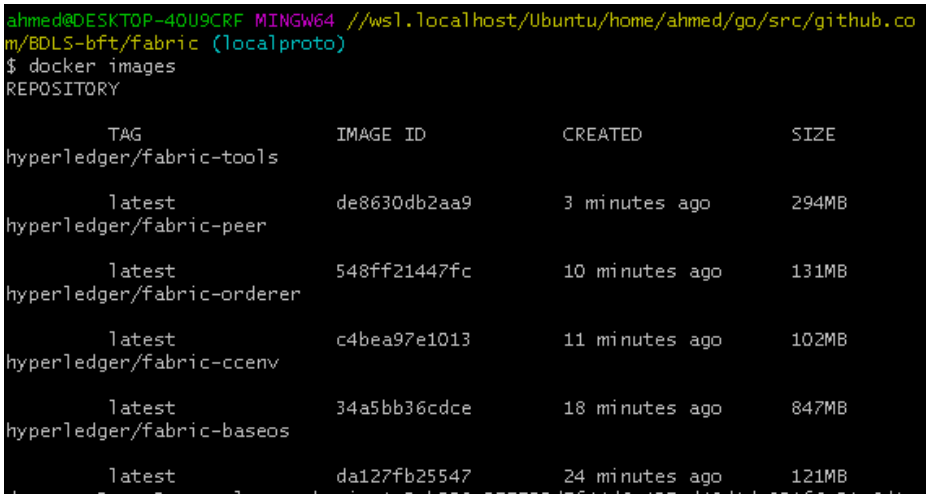


Figure 3.13: Hyperledger Fabric docker images

3.6 Deployment of the Ordering Service

In order to establish an ordering service comprising a set of ordering nodes, commonly referred to as `orderers`, the following steps outline the procedure for creating a BDLS ordering service with four nodes, all affiliated with the same organization. As illustrated in Figure 3.14.

3.6.1 Orderer initialization

The Orderer node initialization and starting can be done through two distinct approaches. The first approach involves utilizing the system channel, which necessitates the presence of the genesis block. This first approach is `deprecated` on September 2023 for the releases after version 2.5. Alternatively, the second approach entails initializing the Orderer without a system channel. This second approach is the only technique for the new Orderer version 3.0, Which is Fabric's latest version of the BDLS integration process.

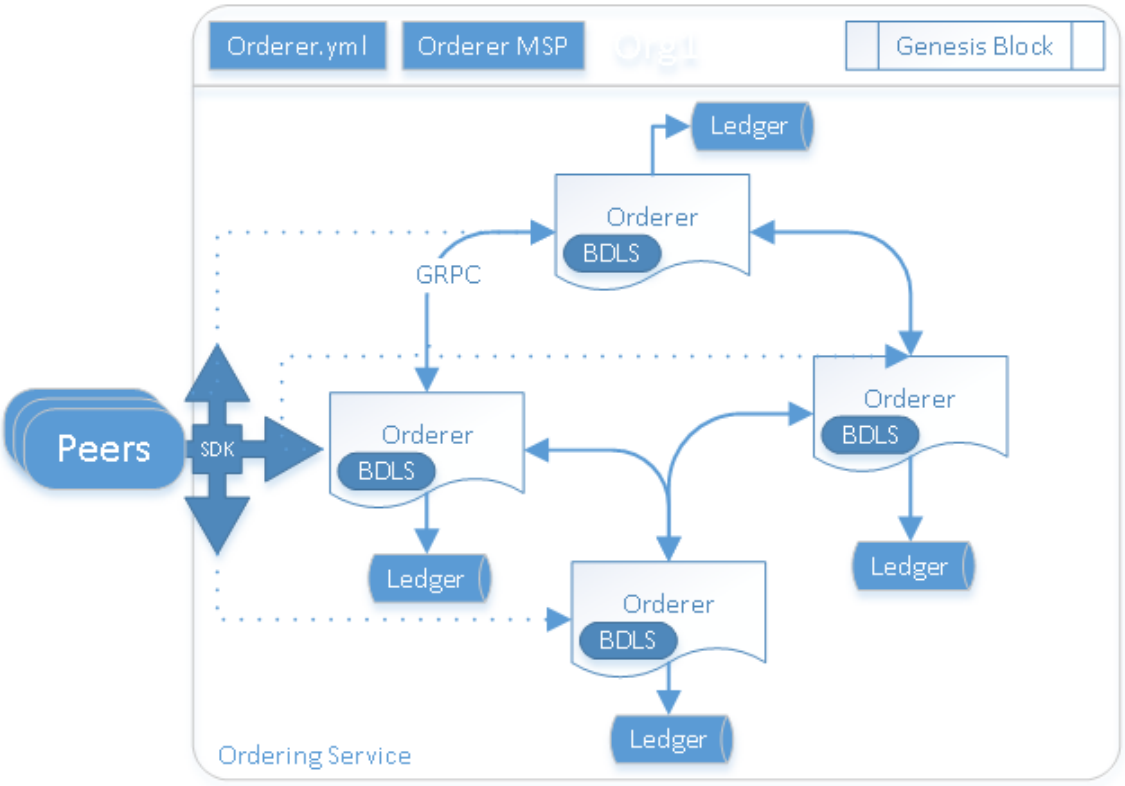


Figure 3.14: BDLS Ordering Service nodes (OSN)

3.6.2 Running the Orderer node

Utilizing the `test-network-nano-bash` [99] project administered by the fabric maintainers, this study investigates the direct execution of the Fabric network from the source code within the fabric-samples project. Clone the *"fabric-samples"* repository at an equivalent organizational directory level as the Fabric. This proximity facilitates the compilation and incorporation of modifications targeting the Orderer node. The procedure involves removing the existing Orderer binary file if it exists within the fabric project's build/bin folder, generating a new Orderer binary file by executing the command:

```
cd fabric/
rm -f build/bin/orderer & make orderer
```

The command line mentioned above deletes the existing Orderer binary and creates

a novel Orderer binary file that the BDLS protocol enables. This new binary file is the execution file of the Orderer node in the sample network. By following these steps, you will successfully obtain the Fabric binaries and the related configuration files, allowing you to proceed with the subsequent steps of the deployment process.

Orderer startup required Genesis Block to run the first time and the runtime characteristic maintained in the `orderer.yaml` file. The `orderer.yaml` is required to run the Orderer node. [100]

To facilitate the accessibility of the Orderer binary executable. Including the Orderer binary location within the operating system's local desk in the environment variable is advisable. This practice ensures that the Orderer binary can be easily recognized and utilized without the necessity of explicitly specifying the full path.

```
export PATH=UNCC-Fabric/bin:$PATH
```

Before initiating the Orderer within a production network, it is imperative to undertake specific preparatory tasks. These tasks encompass creating and organizing essential certificates, generating the genesis block, determining suitable storage options, and configuring the `orderer.yaml` file. These steps ensure the requisite foundational elements and configurations are in place to facilitate the successful deployment and operation of the Orderer in the production environment.

Override the ledger location that is configured in the `orderer.yaml` file by running the shell command:

```
export ORDERER_FILELEDGER_LOCATION=
   $HOME/ledgers/orderer/org1/ledger
```

Change the log level by running the shell command:

```
export ORDERER_GENERAL_LOGLEVEL=debug
# For enabling Advance logging:
export FABRIC_LOGGING_SPEC=debug:cauthdsl,policies
   ,msp,common.configtx,common.channelconfig=info
```

CHAPTER 4: Securing IoT, Edge, and Cloud Systems

4.1 Introduction

This chapter contains material reprinted, with permission, from Ahmed Al Salih, Yongge Wang, Securing the Connected World: Fast and Byzantine Fault Tolerant Protocols for IoT, Edge, and Cloud Systems, 2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW), May 2024. © 2024 IEEE.

In an era marked by rapid technological advancement, the landscape of our world has been fundamentally transformed. Among the most influential innovations is the Internet of Things (IoT), which has emerged as a powerful catalyst for change, particularly in revolutionizing visibility across diverse domains. By furnishing real-time data and insights, IoT has bestowed upon businesses the ability to refine operations and elevate overall efficiency to unprecedented levels. However, amidst this wave of progress, a pressing concern has surfaced: security. This concern is compounded by the decentralized architecture inherent in IoT systems, posing significant challenges to safeguarding devices and the data they generate. Indeed, security has become the paramount issue accompanying the ascent of technology. It is essential to mitigate unauthorized access, prevent data breaches, and avert potential harm.

However, as the demand for real-time data streaming intensifies, a need arises for rapid dissemination protocols to complement stringent security measures. In addition to the challenges mentioned earlier, we consider the most vulnerable industry, which is the supply chain, illuminates further complexities inherent in IoT deployments. The distributed nature of supply chain operations necessitates extensive collaboration among geographically dispersed entities to deliver goods to customers effectively.

However, this very nature renders the network susceptible to potential attacks, heightening the importance of robust security measures. Presently, the industry relies on secure distributed ledger technology systems like Hyperledger Sawtooth, which prioritize Byzantine Fault Tolerance (BFT) behavior through the Practical Byzantine Fault Tolerance (PBFT) protocol. This chapter proposes an alternative approach by integrating Hyperledger Fabric with Byzantine fault tolerance with BDLS, a promising BFT protocol. Our study entails collecting and analyzing data comparing the performance of BDLS and Raft, shedding light on the efficacy of our proposed solution. Additionally, we provide empirical evidence demonstrating the performance disparity between PBFT and BDLS, underscoring our proposed integration's advancements and benefits in enhancing security and efficiency within IoT ecosystems, particularly in supply chain management, smart cities, health departments, and global industries.

In this chapter, we reviewed the Blockchain-IoT in Section 4.2. Then, we discuss the challenges of distributed ledger technology platforms of the IoT and Edge server in section 4.3. We identified the main challenges related to the (security and performance) aspects. In section 4.4 we demonstrate the performance evaluation. Last but not least, we conclude in the last section of this chapter.

4.2 Blockchain-IoT

IoT incorporates physical devices [101], computers, servers, and small objects embedded within a network system [102]. The variety of industries for which IoT devices can be beneficial, such as healthcare, technology, financial, and supply chain industries, is distinguished by the attributes of distributed ledger technology. In other words, the integration of IoT and blockchain technology ensures the data transmitted by the IoT devices are recorded in the right sequence and tamper-prof in the blockchain ecosystem [102].

The distributed ledger technologies are relaid particularly on the consensus algorithm [103] [104] [105] type. We elucidate various consensus types, focusing on

those commonly employed in distributed ledger technology and IoT.

4.2.1 Edge-IoT

The IoT system is a collection of tiny devices with attached sensors that can transmit the data to the system. However, the power of the physical IoT device is limited in the cases of memory, storage, and battery. The edge server technology is responsible for collecting the data from the edge sensors and then pushed to the cloud. The edge server must protect the data from being tampered with using blockchain technology. Many blockchains or distributed ledger technologies (DLT) platforms are integrated into the Edge servers technology. We will review the technologies used in those DLT systems and the review of the challenges in the next section.

4.3 Distributed ledger platforms Challenges of the IoT

The challenges of IoT security and performance have received considerable critical attention [106]. Based on the broad use of IoT Blockchain consensus problem [107]. We categorize the challenges into two main sections: **Security** in 4.3.1 and **Performance** in 4.3.2.

4.3.1 Security

IoT edge characteristics, combined with the presence of devices outside institutions' locations, might reduce protection enforcement. The small size of the devices gives us the ability to distribute them in places that may be public or open, such as fields and farms, or they may be mobile in transportation. The geographic nature of IoT devices makes them vulnerable to thread attacks, which could harm the data and the network. A system that can identify incoming malicious attacks is essential in the IoT ecosystem. The Byzantine Fault Tolerance (BFT) system is the cutting-edge technology in the system that has the feature to identify malicious threads [108].

The Linux Foundation considers Hyperledger Fabric the best framework for enterprise-level technology regarding distributed ledger technology (DLT) in blockchain. How-

ever, Fabric's lack of support for the BFT mechanism makes It not a secure solution for the IoT industries.

Corda [109] is a distributed ledger, unlike a blockchain. Mainly using the Raft protocol, a CFT-type protocol, as we will review in the next section. However, the Corda development team officially declared that Corda with BFT-SMaRt [56] is unstable after a few failed attempts to deploy it [107]. Indicating challenges in achieving stability with this configuration [107].

4.3.2 Performance

The natural behavior of the IoT devices that are used in industries, including smart cities, supply chains, agriculture, and other industries, requires a wide geographic area and a higher number of IoT devices in general. This makes the need for a fault-tolerant discrimination protocol essential. The fault-tolerant type keeps operating in the absence of a certain number of network participants. The number of active nodes that should be active in the distributed system is different from protocol to other, also known as Quorum, as we explained in Chapter One.

However, PBFT [50] is the primarily used protocol or IoT-Blockchain [110] technology that enables the BFT [47] mechanism, or an enhanced version of it such as G-PBFT, [111], Smart-Bft [56], MirBft [112]. Alternatively, several proposed research for securing the IoT technology is about adapting PBFT to support frameworks serving the IoT Systems. [113], [114] The researcher innovated protocols by adding a dynamic leader or multi-leader [112] [56] or adding a reservation phase [113]. Most of the enhanced PBFT studies do not change the message complexity in most of the PBFT inherited protocols as Wang compared in this paper [5] PBFT involves the standard three phases (Per-prepare, Prepare and Commit), starting from the leader proposing to all nodes (Per-prepare) phase, then all nodes send to all nodes(Prepare)phase. Finally, repeating the second phase, in which all nodes send the transaction to all nodes (Commit) phase.

$$2n^2 + n$$

The challenges of security in 4.3.1 and performance in 4.3.2 demonstrated above occurred in the Cloud and Edge servers for the IoT technology since Cloud and Edge servers are servers that hold accurate data for recording, alerts, or further AI modules for analysis, prediction, Etc.

4.3.3 BDLS-IoT edge integration

Selecting the high availability enterprise-level framework for the distributed ledger technology, Hyperledger Fabric. Enhance the performance and security of the IoT technology. We successfully integrated the BDLS BFT protocol with Fabric and evaluated the performance in the following subsections. The integration of BDLS with Fabric based on the Fabric Latest LTS Version 2.5, also with the current main branch, contains the up-to-date upgrades. The traditional IoT implementation transmits to the server or cloud server directly. As shown in Figure4.1.

4.3.4 Edge risk prevention

The different natures of IoT edge systems in Smart cities, supply chains, health, manufacturing, or any industry will be in a similar fundamental IoT-edge architecture diagram, as depicted in Figure 4.2. This addition emphasizes the universality of the IoT-edge architecture across various industries, as depicted in the referenced diagram.

The nature of this work renders the devices susceptible to potential hacking threats. In cases where these devices are situated in public or disparate locations, there exists a risk of system malfunction or inadvertent entry of erroneous data. Denial of Service (DoS) attacks are common by proposing data to the BFT system that puts the system in loops that cannot make decisions.

Our implementation and proposed BDLS protocol is to reject the malicious transition. Start from the last decided block height without putting the system in a Denial

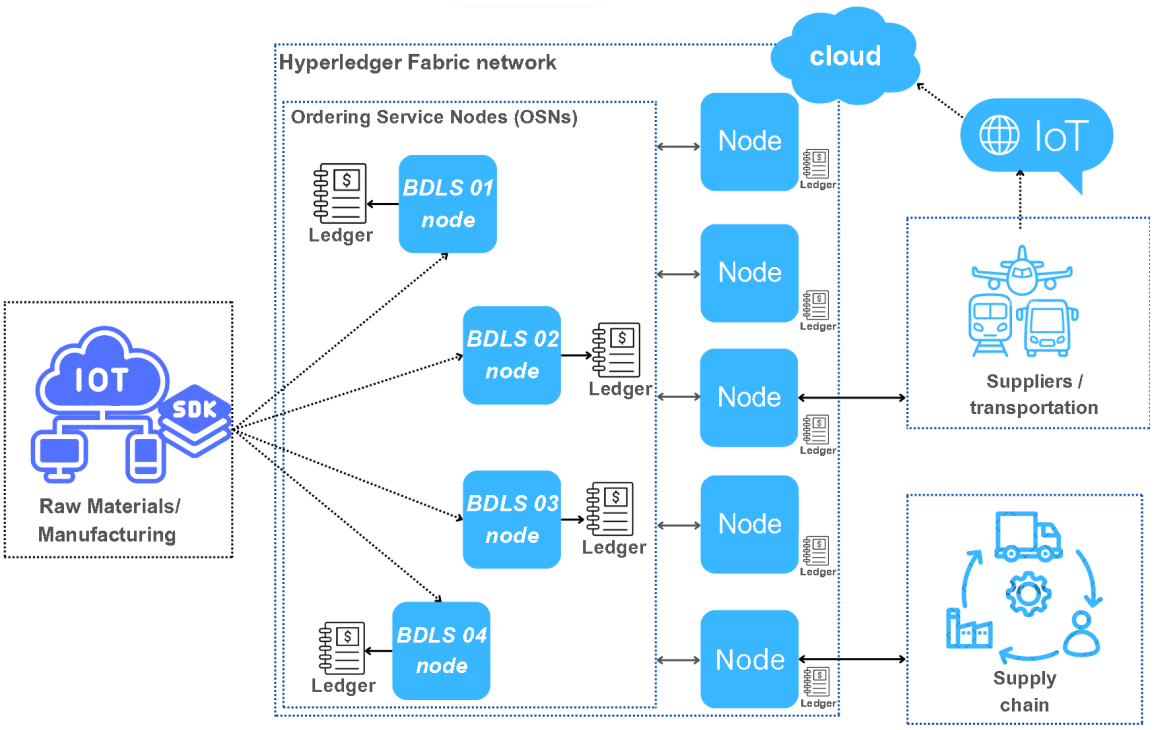


Figure 4.1: IoT SDK connected to Fabric network. © 2024 IEEE.

of Service (DoS) state.

By implementing Hyperledger Fabric-BDLS, we have tailored a Byzantine Fault Tolerance (BFT) solution to ensure the secure processing of data. This adaptation addresses the critical challenge of integrating Edge servers into the system architecture. These servers play a vital role in collecting data from IoT sensors and transmitting it to the cloud. In Figure 4.2, we highlight the significance of the Edge server, which acts as a crucial intermediary. It is essential to note that the Edge server must handle incoming data with caution, as it cannot inherently trust the information it receives. Moreover, leveraging blockchain technology in the cloud offers enhanced efficiency in handling high throughput Transaction Per Second (TPS) rates. This is particularly beneficial for real-time data processing from a diverse array of Edge systems, as shown in Figure 4.2.

In this version, I clarified the role of the Edge server and its significance in the system architecture. Additionally, I emphasized the importance of blockchain tech-

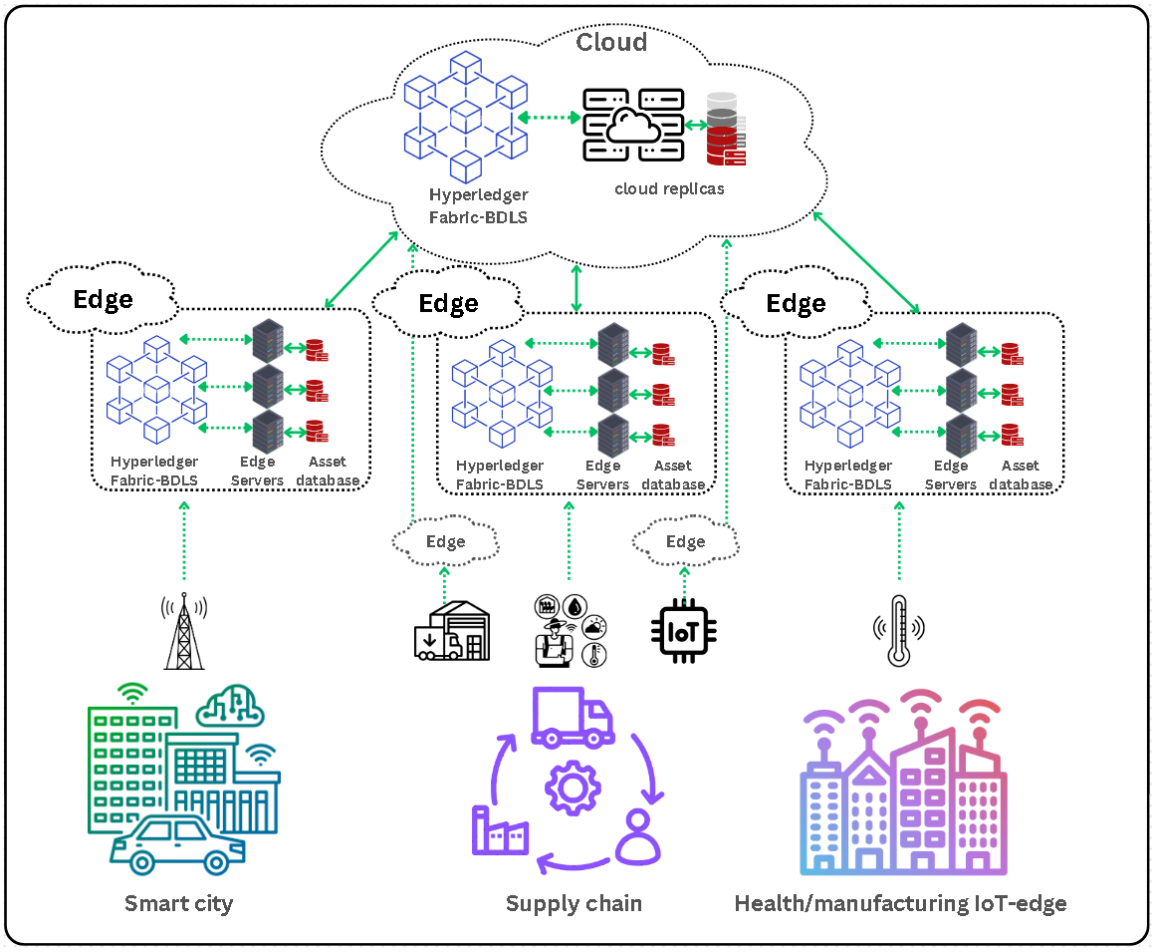


Figure 4.2: IoT-Edge - Fabric-BDLS cloud network. © 2024 IEEE.

nology in improving efficiency and handling high throughput rates, especially when processing data from various Edge systems in real-time.

4.3.5 Cloud risk prevention

Edge servers or individual IoT devices pose a potential threat of malicious attack, capable of inducing a Denial-of-Service (DoS) state on the cloud replica or introducing incorrect or misleading data, thereby impacting the integrity and accuracy of the system's analysis. Safeguarding the cloud servers with a secure and efficient Byzantine Fault Tolerance (BFT) protocol, such as the BDLS protocol proposed by us, presents the optimal solution for ensuring dual-edge security (both at the edge server-side and the cloud side). As shown in Figure 4.2.

4.4 IoT Performance Evaluation

4.4.1 Experimental Setup

We have developed a comprehensive benchmark tool that enables the concurrent execution of multiple clients, simulating the behavior of anticipated IoT-edge devices transmitting data to the system. This benchmark interacts with each BDLS replica within the network, submitting or proposing message data to evaluate their performance under varying conditions. Our testing environment utilizes the Fabric maintainer test platform known as "Test Network Nano bash" [99]. This platform provides the necessary infrastructure for operating the network using the Fabric core source code, facilitating seamless testing and evaluation of our system's performance. The hardware specifications of our system are configured to accommodate average to minimal computing power. This choice is deliberate, as we do not anticipate all the edge servers to be high-performance "super servers" capable of scaling computing power efficiently, as might be found with cloud services (IaaS, PaaS, or SaaS). Our server runs on Ubuntu 22.04.3 LTS with 8GB of RAM and an Intel(R) Core(TM) i5-9400 CPU @2.90 GHz. These specifications were chosen to align with the ex-

pected workload and to ensure optimal performance and scalability during testing. The experimental message configuration is based on a comprehensive review of recent IoT-blockchain research, specifically focusing on experiments that examine the total number of messages per block, network latency, and message size.

4.4.2 Experimental test results

For the evaluation, we identify the consensus protocol competitors. The Raft protocol is at the top of the list. Raft is the official protocol operating Fabric. Also, existing published throughput TPS results recently by the Hyperledger website [115].

Incorporating a batch size of 500 and two message sizes (100 bytes and 1000 bytes) [115]. BDLS has an average latency of 100ms. A minimum of 20ms. The max latency set in the BDLS consensus core is 200ms.

Table 4.1: Benchmark results 100 Byte Asset Size, TX 100K (IoT configuration)

Protocol	Succ	Fail	Max Latency	Min Latency	Throughput (TPS)
Raft	100000	0	2.13	0.18	1153.00
BDLS	100000	0	2.0	0.20	1130.00

The experimental data is collected by running a Fabric test network [99] operating four BDLS Orderer nodes and proposing 100,000 messages. The data size for a single message is 100 bytes. The total time from proposing the first block until writing the last proposed message is 88 seconds with 1135.64 TPS. Using the same network configuration, we ran the Fabric test network operating three Raft Orderer nodes, proposing the same number and sizes of message numbers. The total time from proposing the first block until writing the last proposed message is 86.7 with 1158.75 TPS. The results mentioned are demonstrated in Table 4.1.

The second competitor is the Hyperledger Sawtooth, including the best Performance Scalability published results [116] applying work switching from a serial scheduler to a parallel scheduler. In their experiment and Performance Scalability section,

The throughput has increased by over 30%, with the maximum figure rising from 11.68 tps to 16.37 tps [116].

Note that *Hyperledger Sawtooth* configuration can be adjusted regarding the batch size referred to as Maximum Batches Per Block (MBPB). Similar to Hyperledger Fabric, permitting the user to choose the maximum amount of messages that can be grouped in a batch "MaxMessageCount". The results of the two systems cannot be compared even if we duplicate the last enhanced results for Hyperledger Sawtooth 200 TPS versus Fabric-BDLS 1,130 TPS throughput.

Conclusion

The novel architecture for interfacing with BDLS core functions is distinct, providing enhanced memory efficiency and performance optimization. It specifically utilizes *c.consensus.Update(time.Now())* and direct access to *c.consensus.CurrentState()* within an infinite loop in Hyperledger Fabric, replacing the conventional Go routine implementation. Furthermore, the periodic execution of these functions via a preset ticker timer acts as the heartbeat mechanism for BDLS. This design not only exemplifies an optimal framework for BDLS-Fabric integration, particularly in terms of performance efficiency and implementation efficacy, but it also aligns the interaction with BDLS functions in the *startConsensus* function with Fabric's standard functions, facilitating easier customization.

In conclusion, the integration of Hyperledger Fabric with the BDLS Byzantine fault-tolerant (BFT) consensus protocol presents a compelling solution for enhancing the security and performance of distributed ledger technology, particularly in the context of large-scale networks vulnerable to Byzantine faults. The BDLS protocol, with its optimized message complexity and enhanced efficiency, scalability, and security, offers a promising approach to addressing the challenges faced by traditional BFT consensus mechanisms. Compared to other protocols, such as PBFT family protocols and other non-BFT protocols, BDLS demonstrates superior performance

with fewer communication steps. By integrating BDLS with Hyperledger Fabric, a high-availability enterprise-level framework for distributed ledger technology, we can leverage the resilience of both systems to create a robust and secure environment for transaction processing. This integration is particularly valuable in IoT edge environments, where data integrity and security are paramount. Utilizing Fabric-BDLS integration enables the deployment of secure and efficient IoT solutions. Data collected from sensors at the edge is processed and validated securely before transmission to the cloud. The edge server plays a crucial role in this architecture, ensuring the trustworthiness of incoming data and mitigating potential threats. Overall, the integration of Hyperledger Fabric with the BDLS protocol represents a significant step towards achieving secure and high-performance distributed ledger systems, especially in scenarios where trust among participants cannot be guaranteed, such as large-scale IoT deployments.

CHAPTER 5: Performance Evaluation

5.1 Introduction

Integrating BDLS into the Fabric Orderer architecture aligns closely with the Raft Orderer implementation. It represents the pioneering instance of successful BFT protocol integration within the standardized Fabric Orderer architecture, constituting a novel consensus protocol amalgamation. The recent release of Fabric-BDLS allows system administrators to operate the network by choosing between BDLS Orderer and Raft-type Orderer nodes. BDLS offers the flexibility to coexist with Raft within a unified build, making it an adaptable choice. BDLS-Fabric is a valuable guide for integrating various consensus protocols, facilitated by its transparent Orderer implementation. The parallelism between BDLS and Raft Orderer implementations encompasses protocol initiation, message reception, order orchestration, batching, proposal submission to the core protocol, and the receipt of blocks, all within the same channel. Notably, the implementation within the Fabric core code remains transparent, devoid of concealed logic or migration of implementation details to the protocol code, enhancing the clarity and reliability of the integrated system.

5.2 Experimental Environment

The test network facilitates Fabric's learning process by enabling developers to deploy nodes on their local machines. It also offers a platform for testing smart contracts and applications, allowing developers to validate system functionalities. [95].

The `fabric-samples` project repository [97], which utilizes the Fabric test network [95], has undergone a configuration folder restructuring. The configuration folder has been relocated from the repository's root directory to the `/test-network/configtx`

folder. This updated configuration folder now includes only the `configtx.yaml` file, which contains essential configuration parameters for the network.

The procedure for configuring the environment variable involves assigning a specific key-value pair, where the key is:

$$FABRIC\_CFG\_PATH$$

This has been automated to eliminate the need for manual interaction. Upon executing the `network.sh` file to start the test network, the environment variable will be automatically set, ensuring the correct configuration path is utilized.

```
export FABRIC_CFG_PATH=${PWD}/configtx
```

Build Fabric Docker images and binaries in the fabric root directory

```
make configtxgen configtxlator cryptogen orderer peer osnadmin
   docker
```

Starting from Fabric version 2.3 and above, a new binary command `osnadmin` was released. Administrators can utilize the "`osnadmin channel`" command to execute operations pertaining to channels on an Orderer. These operations include the processes of joining a channel, enumerating the channels linked to a specific Orderer, and deleting a channel are important functionalities [24]. To activate the channel participation API and set up the Admin endpoint for each Orderer, it is necessary to modify the configuration within the `orderer.yaml` file [24].

Two distinct categories of build artifacts are generated during the build process: binaries and Docker images. These artifacts are located within the newly created build folder, which is situated in the root directory of the fabric project. The binaries can be found in the `build/bin/` directory, while the Docker images references are stored in the `build/images/` directory.

Verify the integrity of the build artifacts for the binary, proceed to the build/bin

folder, and examine the list of binary artifacts. The generated docker images can be listed by executing the command ‘‘`docker images`’’ and ensuring that the generated docker images are displayed.

To establish an ordering service comprising a set of ordering nodes, commonly referred to as `orderers`, the following steps outline the procedure for creating a BDLS ordering service with four nodes. Note that the minimum node number to run the BDLS protocol is four. For most of the BFT protocols, including BDLS, the Quorum rule is $3f + 1$.

The transaction submitted by a client peer node will be broadcast to all Orderer nodes throughout the SDK. Affiliated with the same organization. As illustrated earlier in Figure 3.14.

5.2.1 Running the Orderer node

Utilizing the `test-network-nano-bash` [99] project, maintained by the fabric developers and maintainers. In our test, we recommend a direct execution of the Fabric network from the source code within the fabric-samples project. Clone the *“fabric-samples”* repository at an equivalent organizational directory level as the Fabric. This proximity facilitates the compilation and incorporation of modifications targeting the Orderer node. The procedure involves removing the existing Orderer binary file if it exists within the fabric project's build/bin folder, generating a new Orderer binary file by executing the command:

```
cd fabric/
rm -f build/bin/orderer & make orderer
```

The operating system we used for the test evaluation is Ubuntu 22.04.3 LTS. The hardware properties of the system are 8GB of memory, and the processor is Intel(R) Core(TM) i5-9400 CPU @2.90 GHz.

5.2.2 Benchmarking Setup and Tools

The benchmarking tool employed in this study is designed to run multiple clients concurrently. Each client can submit several transactions, and each transaction is created by generating a `common.Envelope` object. Within the payload, the message type is specified in the Header, and the message size is defined in the Data section. The Data section contains plain text, which dynamically updates each message to verify the data written into the block.

Each `Envelope` object is sent to the Order function in the chain to submit the transaction.

5.2.3 Performance Measurement: TPS Calculation

The principle of TPS (Transactions Per Second) calculation for Hyperledger Caliper [117] is to measure the throughput, which is the total number of transactions divided by the total time in seconds [118].

The calculation process begins by setting the start time immediately before calling the function responsible for creating the Envelope object. The end time is recorded after the last message is received and written into the block file. This ensures an accurate measurement of the time taken to process all transactions.

The benchmark measures TPS by determining the number of transactions processed per second.

$$\text{TPS} = \frac{(TotalTxNumber)}{(\text{Tx submission time - block write time})}$$

Where (float64(Tx submission time - block write time)) represents the total time in nanoseconds taken to process the transactions. This approach ensures precise and reliable measurement of the TPS, which is critical for evaluating the performance of different consensus protocols in Hyperledger Fabric.

For example, during our experiments, we recorded the start time once creating the

Tx to be sent and the end time after the last Tx message was written to the block. By applying the above formula, we obtained the TPS values, which provided insights into the efficiency of the consensus protocols under test.

By adopting this method, we can compare the performance of various consensus mechanisms, such as Raft, SmartBFT, and BDLS, under different network conditions.

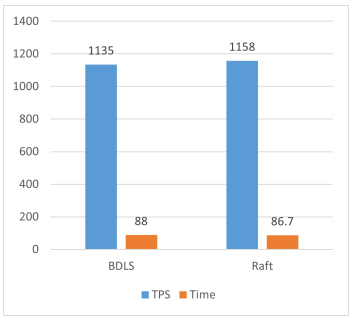
The principle of TPS Calculation for Hyperledger Caliper [117]. is calculated as Throughput. which is the total number of transactions divided by the total time per second [118]. The benchmark measures the Transactions Per Second (TPS) duration by setting the start time once the first transaction is submitted; the end time is set after the last message received is written in the block.

5.3 Experimental Results (BDLS-Raft)

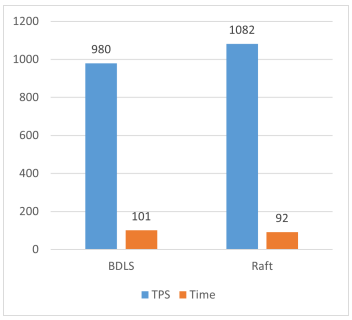
In our performance evaluation, the benchmark is configured based on the latest performance test specifications published by Hyperledger in February 2023 [115]. Incorporating a batch size of 500 and two message sizes (100 bytes and 1000 bytes) [115].

BDLS has an average latency of 100ms. A minimum of 20ms. Raft utilized a Fabric network with three Orderer nodes, while BDLS ran four. The Fabric release operated in the tests is the latest main branch, including Orderer 3.0.0, up to date as of December 2023. The Fabric Orderer configuration used for the benchmark in our experimental assessment is similar to the configuration used by the latest Hyperledger published result [115], as preferred max bytes is 2 MB, the block size is 500, and block cut time is 2s [115]. The evaluation starts by proposing 100,000 messages with a size of 100 bytes to BDLS, which took 1m28 with 1135.64 TPS, compared to Raft with identical message numbers and sizes, which took 1m33 with 1158.75 TPS. As demonstrated in Figure 5.1a.

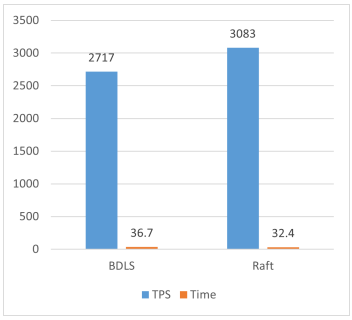
In the second test round, we examined both protocols with a big message size of 1000 bytes. Sending 100,000 messages, the Fabric network that runs four Orderer nodes operates BDLS, a total time of 1m41, which gives 980 TPS to write all the



(a) msg size=100 byte, batch size=500



(b) msg size=1000 byte, batch=500



(c) msg size=100 byte, batch=1,500

Figure 5.1: BDLS vs Raft TPS/Time
Transaction number (txNumber)= 100,000

submitted messages in the ledger blockchain. On the other hand, Raft's total time is 1m32, which is 1082 TPS. As demonstrated in Figure 5.1b.

To simulate the recently published Raft TPS result. We increased the max batch size to 1500, as shown in 5.1c. When Raft gives 3083 TPS, BDLS is 2717

Finally, We evaluate the BDLS integration by proposing a high-volume message of 500,000 messages. The single message size is 1000 bytes. Raft TPS is 1022 in 8m9, while BDLS TPS is 964 in 8m38. As demonstrated in Figure 5.2.

In the observed scenario, the BDLS consensus algorithm has considerably narrowed the gap between the Byzantine Fault Tolerant (BFT) model and the Crash Fault Tolerant (CFT) model. Concerning the Transaction Per Second (TPS) metric, BDLS has achieved a TPS score that closely approximates and nearly matches Raft's.

5.3.1 Experimental (Single node evaluation)

We collect data from each participating node involved in decision-making and evaluate their Transactions Per Second (TPS) for each single node in each consensus algorithm. We conduct two rounds of testing. The transaction number (txNumber) is 100,000 messages submitted to the Fabric network. First, three Orderer nodes operating Raft consensus protocol run. Second, the same message number is submitted to the Fabric network that runs four Orderer nodes operating BDLS consensus

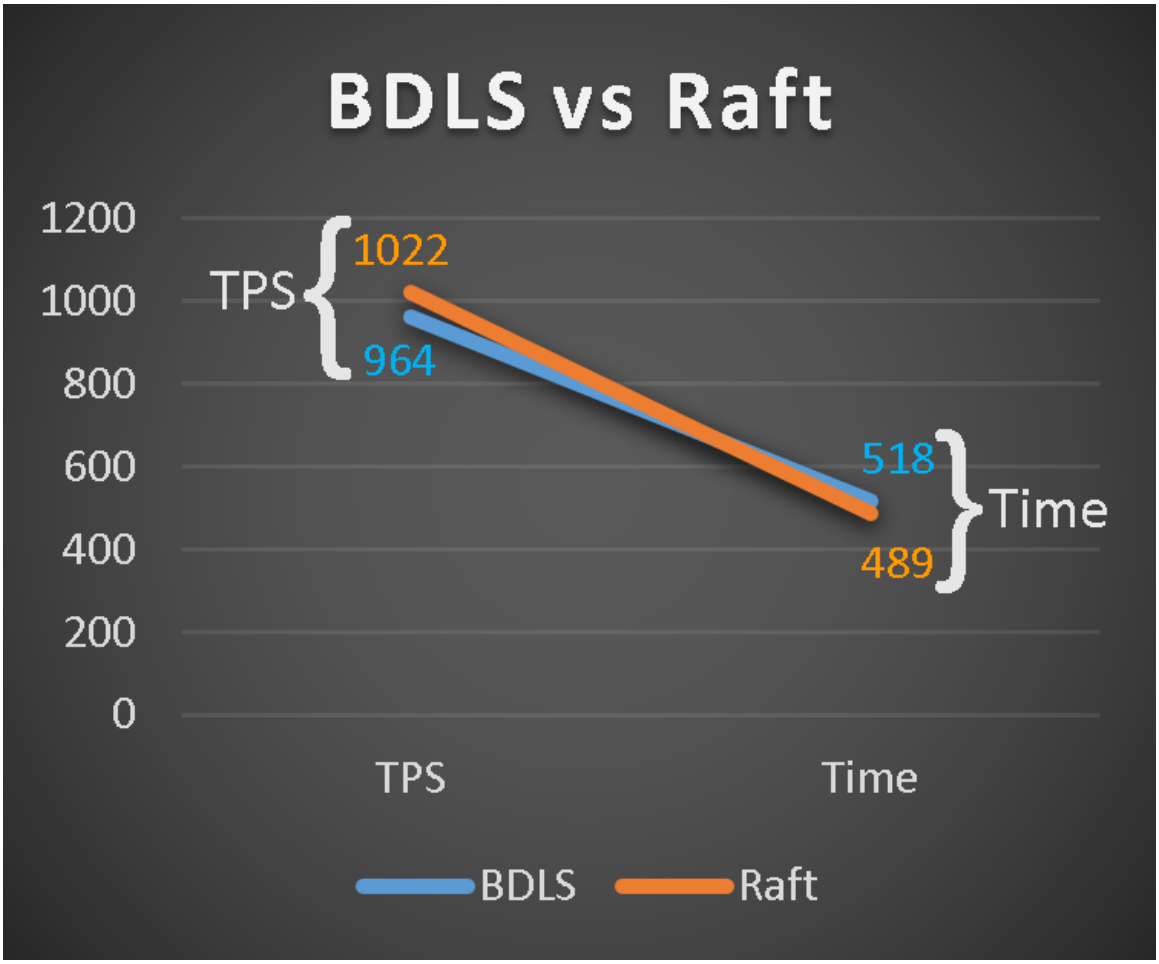


Figure 5.2: BDLS vs Raft 500,000 messages, 1000 byte

protocol. Each message size is 100 bytes. The following results pertain to nodes that achieved consensus successfully. Providing the total time required to write all submitted messages in blocks, calculating the Transactions Per Second (TPS). The results include four BDLS nodes, followed by three Raft nodes test results.

The minimum number of nodes required to run the Raft protocol is three, whereas the BDLS minimum node number is four. In Figure 5.3, each Orderer node displays the (TPS) over time required to conclude the writing of the final block in the node ledger. The four orange circles represent the BDLS nodes. As indicated, a couple of BDLS nodes archived consensus almost simultaneously, as one of the Raft nodes with TPS rang around 1130. In the current case study, the emphasis is on data

Table 5.1: Four BDLS nodes TPS

Protocol replica Node ID	TPS	Time
BDLS 01	1092.583	1 min 31.526 sec
BDLS 02	1115.181	1 min 29.671 sec
BDLS 03	1124.490	1 min 28.929 sec
BDLS 04	1132.310	1 min 28.315 sec

Table 5.2: Three Raft nodes TPS

Protocol replica Node ID	TPS	Time
RAFT 01	1134.266	1 min 28.162 sec
RAFT 02	1145.711	1 min 27.282 sec
RAFT 03	1158.240	1 min 26.337 sec

collection at the single-node level. The data presented for this case research included an estimated one-second latency among nodes to simulate wide network traffic. The test configuration proposes a total TxNumber of 100,000 and a message size of 100 bytes. This plot of the data collected in Figure 5.3 uses the timeline per second.

5.3.2 Experimental setup

The infrastructure for our test uses the operating system Windows 11 Pro Version 22H2, running Windows Subsystem for Linux (WSL2). WSL is a Windows feature that allows users to run a Linux environment on their Windows machine without needing a dual boot or an independent virtual machine. We installed Ubuntu 22.04.3 LTS as the operating system used for the test evaluation.

The Fabric network configuration is based on the latest published performance test results by Hyperledger Foundation [115]. We configure the Fabric network exactly similar to the configuration referenced in the published results [115], which includes the following parameters:

$block\_cut\_time = 2seconds$

$block\_size = 500$

$preferred\_max\_bytes = 2MB.$

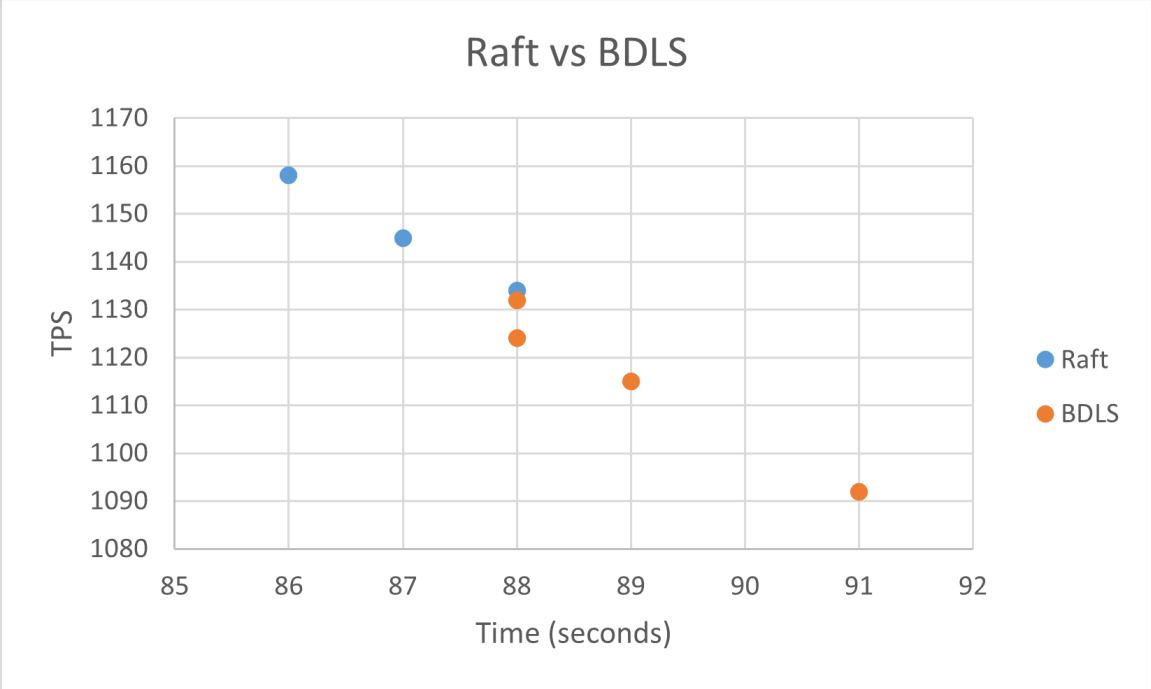


Figure 5.3: (3 Raft / 4 BDLS) nodes TPS/time

The above configuration parameters are adjustable in the *configtx.Yaml* file [24]. Examine the performance with two message sizes of 100 bytes and 1000 bytes.

5.4 Experimental Results

We collect data from the test implemented in three phases, allowing us to present three dataset results in Figure 5.4. We proposed 100,000 messages. The message size is 100 bytes. The transaction per second (TPS) result for BDLS and Raft is represented in blue bars and the time in green bars in Figure 5.4.

BDLS TPS is 1135, whereas The TPS result for RAFT is 1158 in Figure 5.4(a), The time taken to execute the Fabric test network is around 88 seconds using BDLS and 86.7 seconds in the case of RAFT.

The second experimental data collected is represented in Figure 5.4(b). We increased the message size from 100 bytes to 1,000 bytes. The Fabric test network using the BDLS protocol score for TPS is 980. Running the same test on a network using the Raft protocol produces a TPS of 1082.

In the third test shown in Figure 5.4(c), we adjust the batch size equal to 1500, which increased the RAFT TPS to 3083 to simulate the latest published results by Hyperledger Foundation [115], whereas the same test configuration using BDLS produced TPS is 2717.

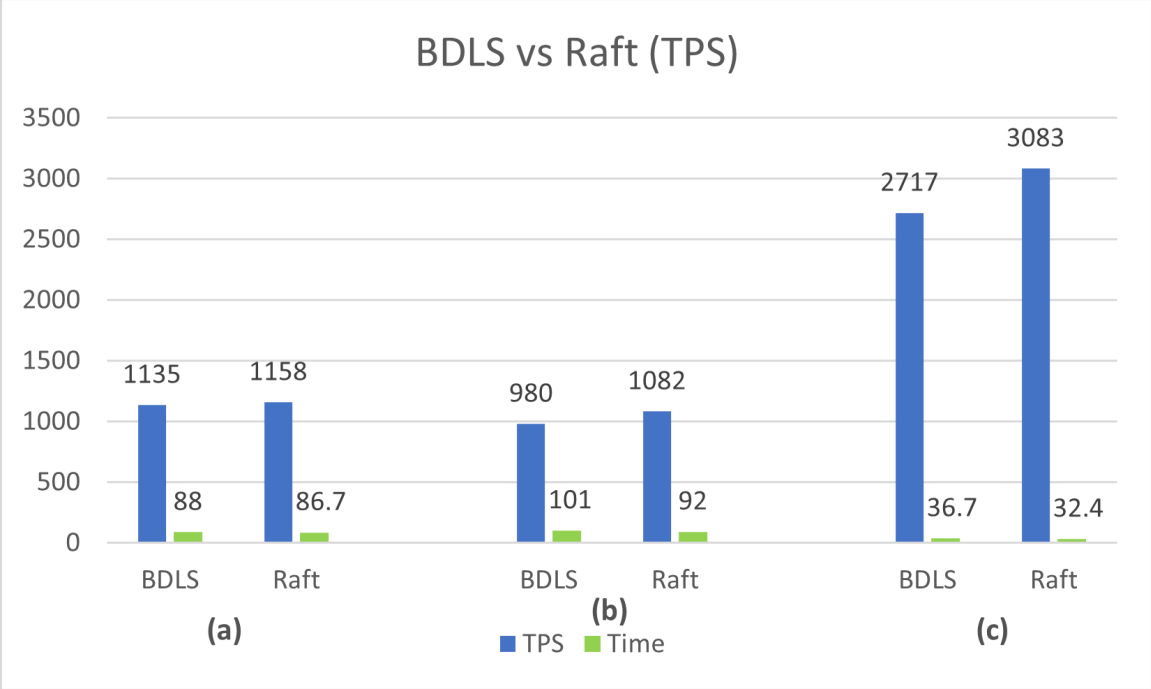


Figure 5.4: BDLS vs Raft TPS/Time.
©2024 ACM. Reprinted with permission from Ahmed Al Salih, Pluggable Consensus in Hyperledger Fabric, BIOTC 2024

We developed the benchmark to calculate individual node throughput in the quorum, which is the minimum required number of nodes participating in the consensus protocol to make a decision. In Figure 5.5, the plot diagram represents two test results, showing the throughput per node and the time that node completed the task in milliseconds for more details per node. The task is Fabric OSNs member of the network participated in achieving consensus and wrote the block. Proposing 100,000 messages with 100 bytes message size and batch size of 500 is the exact test network's configuration by the latest published results from Hyperledger Foundation [115]. The first experimental test shows BDLS node throughput in green. The second through-

put results are for the OSNs run Raft, which is represented in blue, as shown in Figure5.5. Furthermore, We noticed from the experimental results that the throughput of the (raft1) node achieved consensus in 88.162 seconds, a divergent outcome from the throughput of the (bdls4 and bdls3) nodes, which reached throughput at 88.315 and 88.929, respectively.

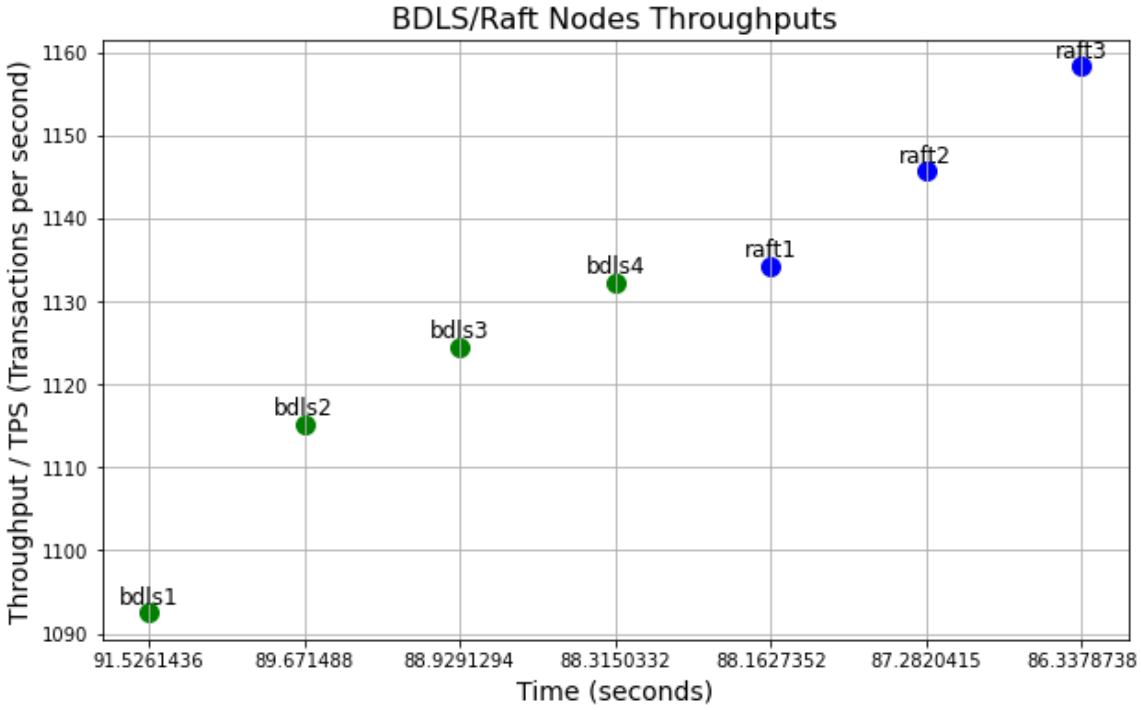


Figure 5.5: BDLS/Raft Nodes Throughput.
©2024 ACM. Reprinted with permission from Ahmed Al Salih, Pluggable Consensus in Hyperledger Fabric, BIOTC 2024

Our research results clearly showed the performance result of the BFT module using the BDLS protocol, bringing us closer than ever to a CFT module for the first time in terms of performance speed or TPS results. Note: Given the absence of publicly available BFT protocols for examination and comparison, we are constrained to utilize the Raft protocol, which is widely recognized as a prevailing and best-practice solution.

5.5 Byzantine Fault Tolerance Experimental

5.5.1 Byzantine Fault Tolerance in BDLS: Node Targeting and Message Divergence

To evaluate Byzantine behavior within the Ordering Service Nodes (OSNs), it is necessary to introduce divergent data, which is subsequently verified by a separate node. In our experimental setup, the test program runs alongside each OSN, enabling us to examine any number of OSNs within the network. There is a need to target specific nodes, which presents certain challenges since each OSN instance is a replica of the first. We address this by assigning different IP addresses, ports, and cryptographic materials to each node.

A key aspect of our test implementation involves embedding a condition in the test client that utilizes the **C.chain** to access the **Order()** function, allowing us to interact with the **c.BDLSID**. By using conditional if statements, we can effectively target specific nodes or a subset of nodes. Additionally, we modify specific messages within the message counter to send targeted messages to particular nodes. This approach enables us to monitor system behavior for deadlock conditions and examine the resulting block messages.

As shown in Listing 5.1, this experiment specifically targets BDLS node 2. The client sends a divergent message, where the message with ID 500 is proposed to this node with modifications. As a result, the modified message is excluded from the final block, and the system instead incorporates the message with ID 500 from the other three honest BDLS nodes.

```
92 func (c *Chain) TestOrderClient4(wg *sync.WaitGroup) {
93   c.Logger.Infof("From Client %v", 4)
94   for i := 0; i < 10000; i++ {
95     if c.bdlsId == 2 && i == 500 {
96       env := &common.Envelope{
```

```
97          Payload: protoutil.MarshalOrPanic(&common.Payload{
98            Header: &common.Header{ChannelHeader:
    protoutil.MarshalOrPanic(&common.ChannelHeader{Type:
    int32(common.HeaderType_MESSAGE), ChannelId: c.Channel})},
99            Data:    []byte(fmt.Sprintf("TEST_BAD-data-Client-4-%v",
    i)),
100         }),
101       }
102       c.Order(env, uint64(0))
103     } else {
104       env := &common.Envelope{
105         Payload: protoutil.MarshalOrPanic(&common.Payload{
106           Header: &common.Header{ChannelHeader:
    protoutil.MarshalOrPanic(&common.ChannelHeader{Type:
    int32(common.HeaderType_MESSAGE), ChannelId: c.Channel})},
107           Data:
    []byte(fmt.Sprintf("TEST_MESSAGE-UNCC-Client-4-%v", i)),
108         }),
109       }
110       c.Order(env, uint64(0))
111     }
112   }
113   wg.Done()
114 }
```

Listing 5.1: Byzantine Behavior Experimental

5.5.2 Byzantine Fault Tolerance Experimental Summery

This experiment demonstrates the Byzantine fault tolerance of the BDLS protocol within Hyperledger Fabric. The test program, implemented in the function **TestOrderClient4**, simulates client behavior by sending 10,000 messages to the network. The program includes a condition that, when the BDLS ID is 2 and the mes-

sage ID is 500, the client sends a modified message **("TEST_BAD-data-Client-4-500")**. The remaining messages follow the standard format **("TEST_MESSAGE-UNCC-Client-4-x")**, where **x** is the message number. The experimental results confirm that the system successfully rejects the divergent message sent by the compromised node and instead utilizes the honest messages proposed by the other nodes. This behavior validates the ability of BDLS to tolerate Byzantine faults by ensuring that the consensus process remains correct despite the presence of faulty or malicious nodes.

5.6 Fabric BFT Experimental (BDLS vs Raft vs SmartBft)

The operating system used for the experimental evaluation is Ubuntu 22.04.3 LTS. The hardware properties of the system are 8GB of memory.

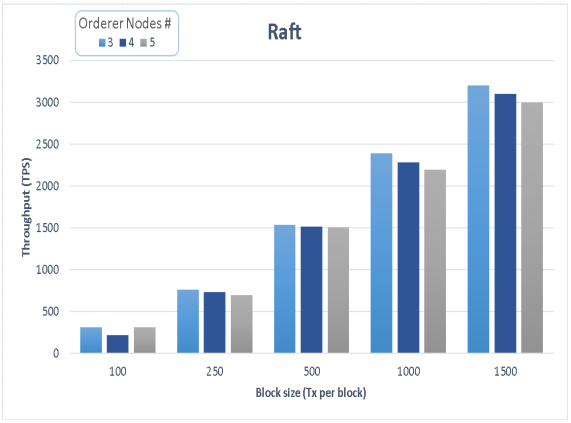
The expected delay is a common property for a distributed system protocol, known as latency. we set the latency for BDLS based on the average expected. The latency to the followers currently stands at an average of 100ms, with a minimum of 20ms recorded in NYC and a maximum of 200ms in Tokyo.

The experimental setup involves comparing the performance of the BDLS, Raft, and SmartBFT protocols in terms of Transactions Per Second (TPS) across different clusters and varying transactions per block. The results are represented in graphs as Figure 5.6 a, b, and c.

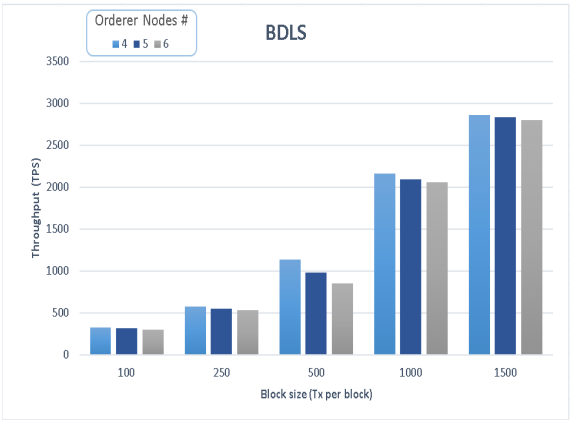
The Fabric Orderer configuration used for the benchmark in this experimental is exactly the same data matrix used by the latest Hyperledger published result [115]. The Fabric release operated in the tests is the latest main branch, including Orderer 3.0.

5.6.1 1. BDLS Performance Assessment

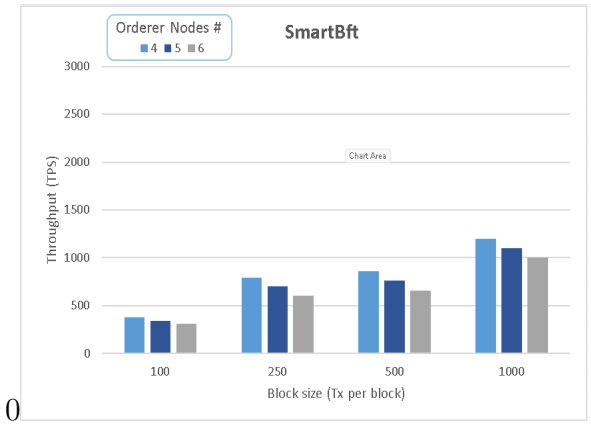
The BDLS protocol, designed as a Byzantine Fault Tolerant (BFT) solution, showed remarkable performance in the experiments that were conducted. In various config-



(a) Raft Experimental



(b) BDLS Experimental



(c) SmartBft Experimental(HLF Orderer 3.0)

Figure 5.6: Throughput (TPS) for Hyperledger Fabric network running: BDLS vs Raft vs Fabric Orderer v3.0 (smartBft). © 2024 IEEE.

urations, including clusters of 4, 5, and 6 nodes, BDLS achieves high TPS values across different transaction loads. For instance, with 1500 transactions per block, BDLS achieves up to 2860 TPS in a 4-node cluster, 2830 TPS in a 5-node cluster, and 2800 TPS in a 6-node cluster. These results indicate that BDLS can efficiently handle high transaction throughput while maintaining the robustness required for BFT systems.

When comparing the performance of BDLS to Raft, as shown in Figure 5.6b and Figure 5.6a, BDLS achieves approximately 89.38% to 93.33% of Raft's transactions per second (TPS) across various cluster configurations. These results indicate that while BDLS introduces Byzantine Fault Tolerance (BFT) capabilities, it maintains a throughput that is closely comparable to that of Raft, even in different cluster setups.

5.6.2 2. SmartBft/Fabric 3.0 BFT Performance Assessment

Our experimental findings regarding SmartBFT closely align with the test results published in [54] by the protocol developers: "In a LAN, BFT-OS achieves 20% the performance of a Raft protocol (2,500 vs. 13,000 tx/sec, respectively in a WAN, SmartBft achieves 40% the performance of a Raft protocol (1,200 vs. 3,000 tx/sec, respectively." [12] [54].

When compared to Raft's performance, SmartBFT in Figure 5.6c achieves approximately 33.33% to 38.71% of Raft's TPS in Figure 5.6a, across different clusters setup.

5.7 Experimental Summery

This section presents a comparative evaluation between BDLS and SmartBFT/-Fabric 3.0, two Byzantine Fault Tolerant (BFT) protocols integrated into Hyperledger Fabric. As the recently proposed beta solution, BDLS has been specifically designed to enhance the ordering service within Fabric by introducing BFT capabilities, offering a more resilient and high-performance alternative to existing consensus mechanisms.

The experimental results demonstrate that BDLS is a highly efficient BFT solution for Hyperledger Fabric's ordering service, delivering both robustness and high throughput. Its performance closely rivals that of Raft, a consensus algorithm traditionally known for its efficiency but not designed for Byzantine Fault Tolerance. Despite Raft's non-BFT nature, BDLS manages to achieve comparable throughput, making it a strong candidate for environments where both high performance and fault tolerance are critical [119].

On the other hand, SmartBFT, although effective in providing BFT properties, exhibits lower transactions per second (TPS) compared to BDLS. This disparity highlights BDLS's superior performance, especially in high-throughput environments where maintaining efficiency under Byzantine conditions is essential. Figure 5.7 illustrates the performance comparison, showing BDLS's clear advantage over SmartBFT in terms of TPS.

The inclusion of BDLS into Fabric marks a significant step forward in the evolution of BFT protocols within the platform, offering a robust and scalable solution that meets the demands of modern distributed ledger systems.

Discussion

The experimental results clearly demonstrate that BDLS offers a substantial performance advantage over SmartBFT, with transaction per second (TPS) rates nearly on par with Raft. Specifically, BDLS's TPS values are only slightly lower than those of Raft, making it a highly efficient Byzantine Fault Tolerant (BFT) solution. This efficiency is particularly significant for Hyperledger Fabric's ordering service, where both high throughput and robustness are essential.

The superior performance of BDLS over SmartBFT can be attributed to its optimized consensus mechanism, which effectively handles transaction validation and block generation while ensuring Byzantine Fault Tolerance. BDLS manages to achieve a balance between performance and fault tolerance, optimizing processes to maintain

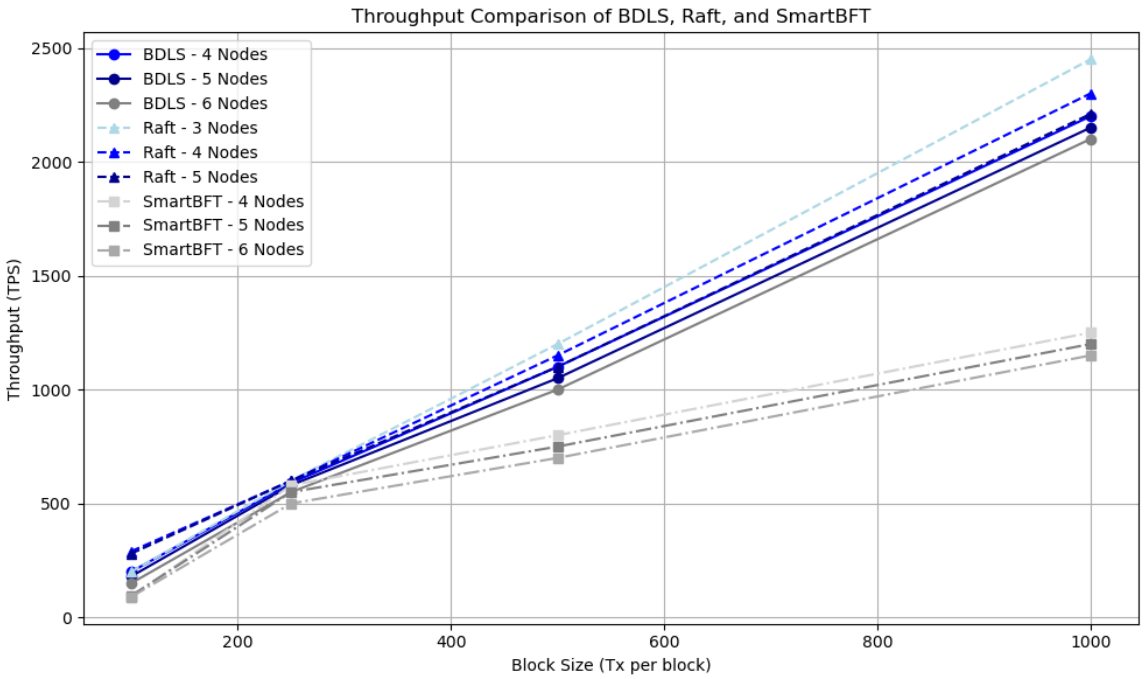


Figure 5.7: Throughput Comparison of BDLS, Raft, and SmartBFT. © 2024 IEEE.

high TPS rates. In contrast, Raft, although not a BFT solution, excels in environments where Byzantine Fault Tolerance requirements are less stringent, thereby maintaining higher TPS under similar conditions. Despite these promising results, there is still considerable room for enhancement. One potential improvement is transitioning the communication among BDLS nodes from using TCP connections to gRPC services. gRPC, with its advantages in performance and scalability, could further enhance the efficiency and reliability of BDLS's communication protocol.

In summary, our experimental results demonstrate the effectiveness of integrating the BDLS protocol into Hyperledger Fabric.

Furthermore, the integration of BDLS with Hyperledger Fabric shows significant potential for future growth. This integration is hosted as an open-source project in the Hyperledger Fabric lab, inviting contributions from the community to further enhance and develop this promising protocol.

CHAPTER 6: Fabric-BDLS Architecture

6.1 Fabric-BDLS Network System Architecture

The *BDLS-Fabric architecture* demos the end-to-end system architecture and transaction flow. This model integrates BDLS consensus algorithm within the Hyperledger Fabric framework. This architecture leverages the modular and scalable design of Hyperledger Fabric, introducing enhanced fault tolerance and security mechanisms essential for enterprise-grade blockchain systems.

6.1.1 Key Components

1. **Certificate Authority (CA):** The CA server issues digital certificates that authenticate and verify the identities of different network participants, such as peers, ordering nodes, and client applications. These certificates are essential for establishing trust within the network and enforcing cryptographic security protocols. The *Membership Service Provider (MSP)* manages these identities so that only authorized participants can engage in the transaction processes.

2. **Software Development Kit (SDK):** The SDK serves as the primary interface between client applications and the blockchain network. Through the SDK, developers and users can invoke chaincode (smart contracts), propose transactions, query the ledger, and submit transactions to peers and the ordering system. The SDK abstracts the underlying complexity of the blockchain network, allowing seamless interaction with various components of the BDLS-Fabric system.

3. **Peer Nodes:** Peer nodes are central to the operation of the BDLS-Fabric network, functioning in two primary roles: *Endorser* and *Committer*.

- **Endorser Peers** execute chaincode and validate transaction proposals against the current data state in the ledger. These peers ensure that transactions conform to the defined business logic before they are submitted to the Orderer.
- **Committer Peers** receives ordered blocks from the Orderer and commits the block in the ledger database, updating the blockchain state. The distinction between endorsing and committing peers allows for scalable transaction processing, where different peers handle validation and ledger updates.

4. **Ordering Service (BDLS Consensus Algorithm):** The ordering service guarantees the correct sequencing of transactions and packaging them into blocks. The BDLS algorithm, a Byzantine Fault-Tolerant (BFT) consensus mechanism, allows the system to tolerate faulty or malicious nodes while maintaining the integrity of the transaction ordering process. The presence of multiple ordering nodes (denoted as *BDLS 01, BDLS 02, BDLS 03, BDLS 04* in the diagram) ensures that the ordering process remains decentralized, resilient, and resistant to a range of faults, including node failures and adversarial behavior.

5. **Ledger and Chaincode:** Each peer node hosts a copy of the *ledger*, which contains the complete history of the transactions that are executed on the distributed system network, and the *chaincode*, or the smart contract logic. The ledger is updated when new blocks are committed, ensuring that all peers maintain a consistent view of the blockchain. Chaincode execution is initiated by client applications through the SDK, and its results are validated by endorsing peers before being submitted to the ordering service for final commitment.

In summary, BDLS-Fabric's architecture combines Hyperledger Fabric's strengths in modularity and permissioned access with BDLS's fault-tolerant consensus, pro-

viding a scalable, secure, and robust solution for distributed ledger applications in high-trust environments.

The overall architectural view of the Hyperledger Fabric-BDLS network is depicted in Figure 6.1. The process commences with the client initiating a transaction to the peer node that is accessible to them. The peer node then validates the transaction, which is subsequently proposed to all ordering service nodes by the Software Development Kit (SDK). The ordering service is composed of a minimum of four BDLS nodes.

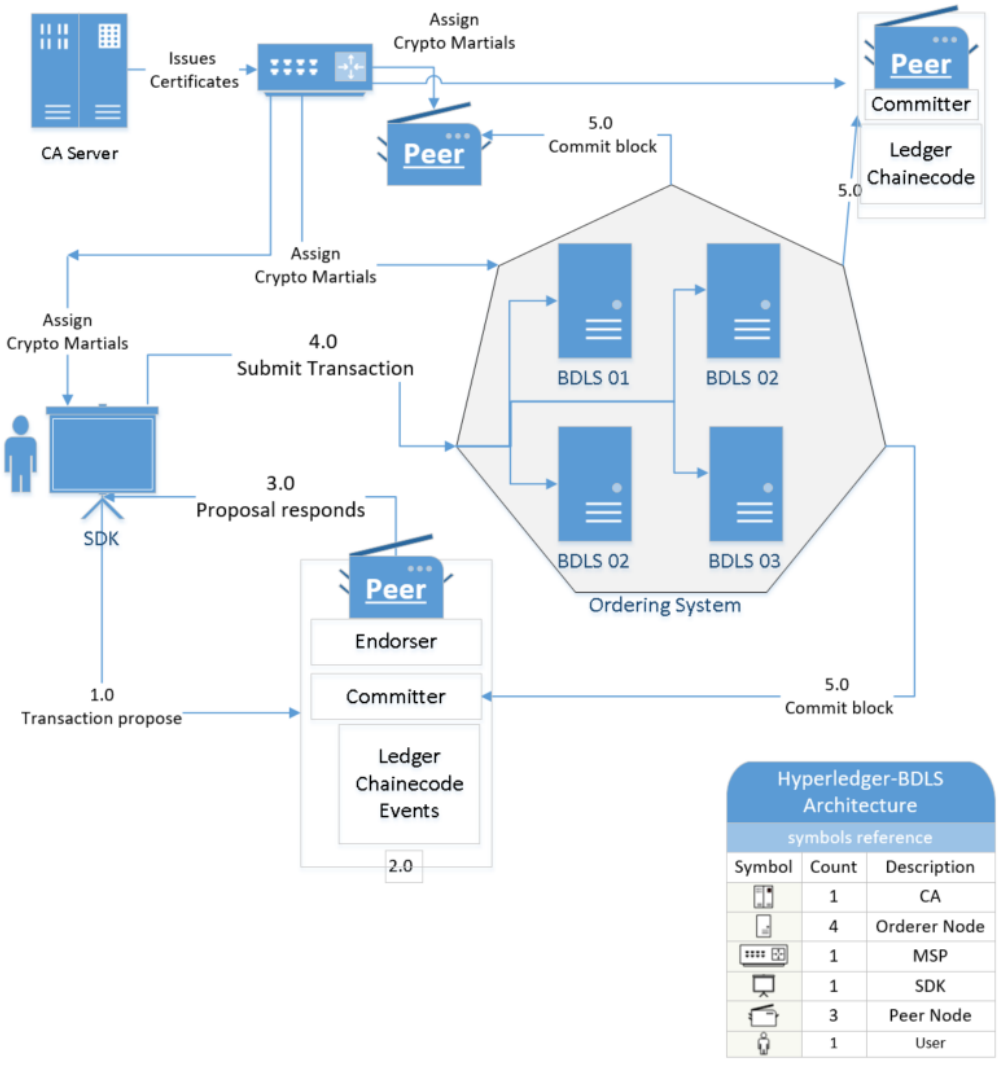


Figure 6.1: BDLS architecture in the complete Fabric system, © 2024 IEEE.

6.1.2 Fabric-BDLS Integration Architecture

The architecture of BDLS integrated with Fabric 3.0, depicted in 6.2, illustrates the data flow from the SDK to the block being written and broadcasted to other orderer nodes.

1. SDK Interaction: The process begins when the SDK initiates a transaction (TX) and submits it to the orderer node via "Order TX," where the transaction is queued for processing.

2. Ordering and Pending: The orderer node batches and orders the transactions, moving them to propose block for consensus.

3. Proposing and BDLS Invocation: After ordering, the transactions are proposed to the consensus mechanism, invoking the BDLS protocol to coordinate with other nodes.

4. Consensus via TCP Agent: The **TCP Agent** manages communication between BDLS nodes, ensuring synchronized state updates and decision-making during consensus.

5. BDLS State Updates and Decision: Nodes exchange information via the **comm** component, updating the BDLS state and confirming the correct block height and transaction state.

6. Block Validation and Writing: After consensus, the validated block undergoes state validation, comparison, and latency adjustments before being finalized and moved to the **Write Block** phase.

7. Block Writing and Local Storage: The block is committed to the orderer node's ledger, ensuring secure storage and system updates.

8. Broadcast to Other Nodes: Through **gRPC**, the finalized block is broadcast to other orderer nodes, with the **comm** module ensuring all BDLS nodes are synchronized with the latest state.

This flow guarantees the ordered, validated, and secure management of transactions

using BDLS as the consensus protocol, ensuring system consistency and performance.

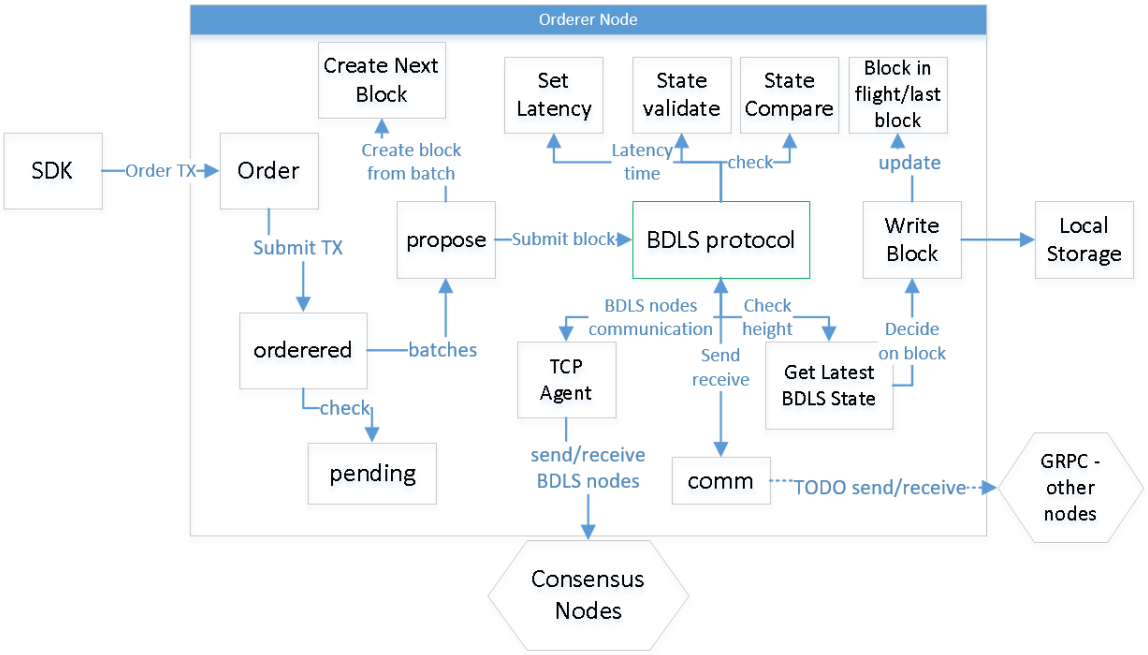


Figure 6.2: Hyperledger Fabric Ordering Service BDLS Architecture

6.2 Fabric BDLS Transaction Flow in the (OSN)

The transaction flow in the BDLS-Fabric architecture involves a series of steps that ensure both the correctness of the transaction and to be written in the ledger (blockchain system). As illustrated in Figure 6.3

In the Fabric BDLS ordering service, client transactions are encapsulated in an envelope object before being submitted to the ordering service. This process begins when transactions are passed to the **Order()** function, which subsequently invokes the **Submit()** function.

Within the **Submit()** function, the envelope is encapsulated as a **Payload** key-value pair inside a **SubmitRequest** object, which is then sent to the **submitC** channel. Each Ordering Service Node (OSN) includes a Goroutine executing the **run()** function, which continuously listens to a set of channels, awaiting incoming data to trigger the appropriate logic.

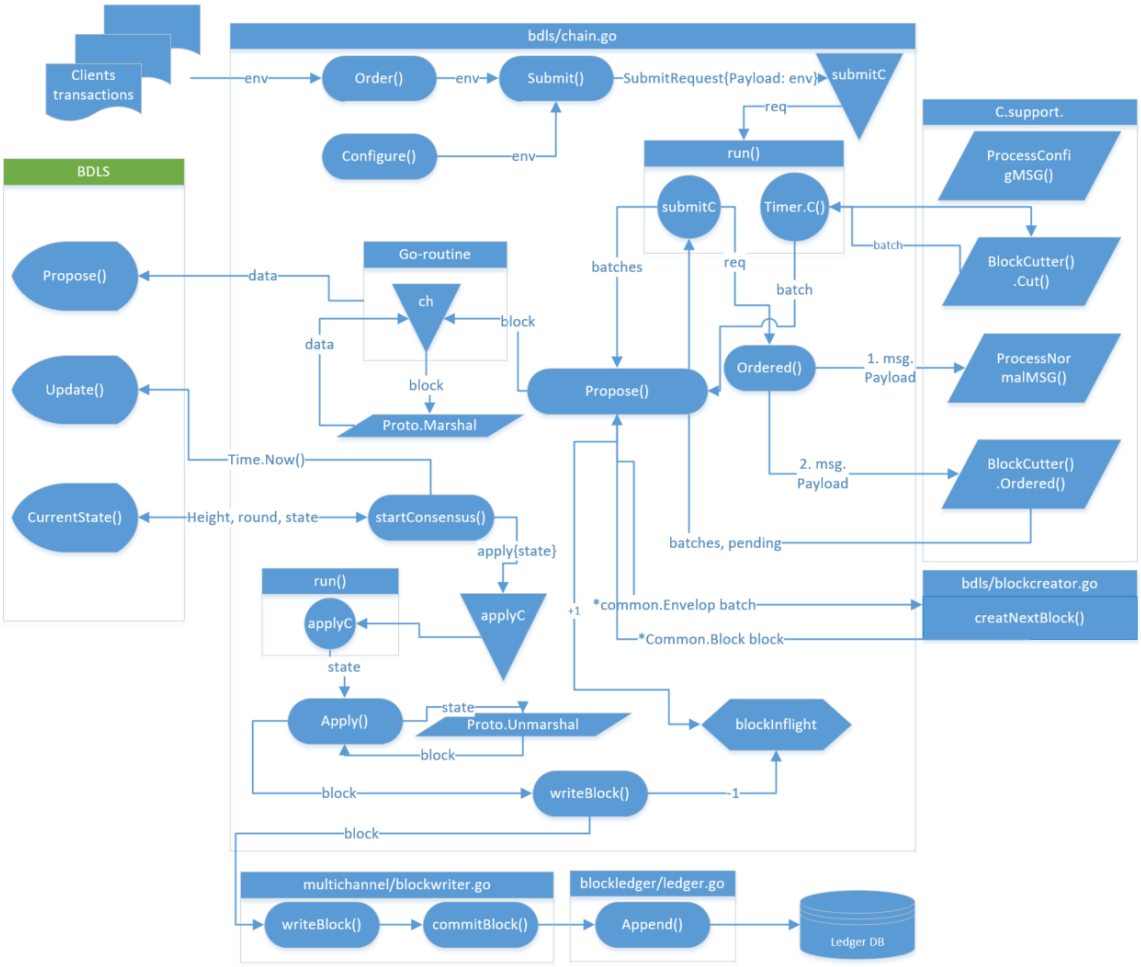


Figure 6.3: Transaction flow in Fabric-Bdls Orderer node

Upon receiving a transaction in the **submitC** channel, it is passed to the Ordered() function. Since this research focuses on normal transactions, the **Ordered()** function validates the message type and forwards it to the **ProcessNormalMSG()** function within the c.support. The **ProcessNormalMSG()** function checks the message's validity against the current system configuration.

The validated message is then sent to the **BlockCutter.Ordered()** function, which handles two key steps: ordering the transactions and batching them into blocks. The function returns two values: the first is the set of transaction batches, and the second is a boolean indicating whether there are any pending transactions. These values are returned when **Ordered()** is called within the **run()** function.

Once the batches are received, they are forwarded to the **Propose()** function along with a channel **(CH)** and a block creator context **(BC)**. The **Propose()** function loops through the batches, treating each as an equivalent to a block. Each block is marshaled, converting it into binary format, and then sent via the **CH** channel. At this point, the block has entered the BDLS protocol library through this channel.

Additionally, a timer **(Timer.C())** is used to ensure that transactions are processed promptly, even if only a single small transaction is submitted. For example, if a timer is configured for 2 seconds, a block will be created within 2 seconds of submission. When the timer expires, it triggers the **support.BlockCutter().Cut()** function, which creates a block using the data from the currently received transaction. This process sends the batch to the **Propose()** function for block creation, completing the first phase of the transaction lifecycle in the OSN.

The second phase involves verifying if the block reaches consensus. The **startConsensus()** function sets up a timer ticker that triggers every 20 milliseconds, simulating a heartbeat for the BDLS node. At each heartbeat, the BDLS library's **Update()** function is called with the current time, followed by a call to the **CurrentState()** function. This function returns the height, round, and state of the blockchain. The height returned by BDLS is compared to the current Fabric system's height. If the BDLS height is greater, it indicates that consensus has been achieved on the block state, and the proposed block is confirmed.

The third and final step involves writing the block to the ledger database. The state is sent to the **applyC** channel, encapsulated within an **Apply** structure. Upon receiving the state, the **applyC** channel triggers the **Apply()** function, which unmarshals the state to retrieve the original block. This block is then sent to the **writeBlock()** function.

The **writeBlock()** function passes the block to the multi-channel package in the **blockwriter.go** file. The block is then written to the ledger through the

c.support.writeBlock() function, followed by a call to the **commitBlock()** function, which appends the block to the ledger using the **Append()** function in the **ledger.go** file.

CHAPTER 7: CONCLUSIONS

In conclusion, integrating Hyperledger Fabric with the BDLS Byzantine fault-tolerant (BFT) consensus protocol presents a compelling solution for enhancing the security and performance of distributed ledger technology, particularly in large-scale networks vulnerable to Byzantine faults. The BDLS protocol, with its optimized message complexity and enhanced efficiency, scalability, and security, offers a promising approach to addressing the challenges faced by traditional BFT consensus mechanisms. Compared to other protocols, such as PBFT family protocols and other non-BFT protocols, BDLS demonstrates superior performance with fewer communication steps.

In this research, we have provided an in-depth exploration of the Fabric ordering service, the integration points of the Raft protocol, and an analysis of the Fabric message life cycle. This comprehensive discussion lays the groundwork for integrating any protocol into the Fabric framework. We used the integration of the BDLS protocol as a practical example to demonstrate the application of a Byzantine Fault Tolerance (BFT) module within Fabric.

Our research reveals that BDLS shows exceptional performance, comparing favorably with the RAFT protocol, whether handling low or high transaction volumes and maintaining this superiority across various message sizes. This enhancement in performance firmly positions BDLS as a viable and practical choice for systems that require efficient, reliable, and secure transaction processing capabilities.

Moreover, the notable performance of the BDLS BFT module represents a significant advancement in our understanding and implementation of decentralized networks. More than ever, it narrows the gap between Byzantine Fault Tolerance (BFT)

and Crash Fault Tolerance (CFT) modules, particularly regarding transaction speed and TPS efficiency. Consequently, BDLS stands out as the optimal BFT module choice for decentralized networks, offering a combination of high efficiency, scalability, and robust transaction processing.

By integrating BDLS with Hyperledger Fabric, a high-availability enterprise-level framework for distributed ledger technology, we can leverage the resilience of both systems to create a robust and secure environment for transaction processing. This integration shows valuable advantages in IoT edge environments, where data integrity and security are paramount. Utilizing Fabric-BDLS framework enables the deployment of secure and efficient IoT solutions, where data collected from sensors at the edge is processed and validated securely before transmission to the cloud. The edge server plays a crucial role in this architecture, ensuring the trustworthiness of incoming data and mitigating potential threats. Overall, the integration of Hyperledger Fabric with the BDLS protocol represents a significant step towards achieving secure and high-performance distributed ledger systems, especially in scenarios where trust among participants cannot be guaranteed, such as large-scale IoT deployments.

Our research underscores the limitations of Hyperledger Fabric's newly integrated SmartBFT protocol, particularly its high message complexity and scalability issues. The BDLS protocol, in contrast, demonstrates superior scalability and efficiency, maintaining linear message complexity regardless of network size. Our comparative analysis reveals that BDLS-based Fabric achieves comparable throughput to Raft-based Fabric while offering enhanced Byzantine fault tolerance. This makes BDLS a promising alternative for large-scale decentralized applications across various industries. Future work should focus on further optimizing BDLS integration and exploring its performance in diverse real-world scenarios to validate its practical benefits. Our findings advocate for broader adoption of BDLS in Hyperledger Fabric to achieve more robust and scalable blockchain solutions.

REFERENCES

[1] F. B. Insights, "Blockchain market size, share covid-19 impact analysis." https://www.fortunebusinessinsights.com/industry-reports/blockchain-market-100072, 2022.

[2] MarketsandMarkets, "Blockchain market - global forecast to 2030." https://www.marketsandmarkets.com/Market-Reports/blockchain-technology-market-90100890.html, 2022.

[3] A. M. Research, "Supply chain blockchain market analysis." https://www.alliedmarketresearch.com/blockchain-supply-chain-market, 2023.

[4] ResearchAndMarkets, "Global blockchain in healthcare market report." https://www.researchandmarkets.com/reports/5238457/blockchain-in-healthcare-market-global-industry, 2024.

[5] Y. Wang, "Byzantine fault tolerance for distributed ledgers revisited," *Distributed Ledger Technologies: Research and Practice*, vol. 1, no. 1, pp. 1–26, 2022.

[6] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pp. 305–319, 2014.

[7] "Blockchain statistics & facts that will make you think." https://fortunly.com/statistics/blockchain-statistics/, 2021.

[8] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system-satoshi nakamoto-the bitcoin whitepaper." https://bitcoin.org/bitcoin.pdf, 2009.

[9] M. d. Castillo, "Blockchain 50 2021. forbes." https://www.forbes.com/sites/michaeldelcastillo/2021/02/02/blockchain-50/?sh=3620834f231c, 2021.

[10] â. â. O. S. B. Technologies, "â hyperledger," Aug. 2021.

[11] C. Cachin *et al.*, "Architecture of the hyperledger blockchain fabric," in *Workshop on distributed cryptocurrencies and consensus ledgers*, vol. 310, pp. 1–4, Chicago, IL, 2016.

[12] J. Sousa, A. Bessani, and M. Vukolic, "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, 2018.

[13] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, pp. 1–15, 2018.

[14] â. T. Projects, "â hyperledger projects 2021," Aug. 2021.

[15] K. Wüst and A. Gervais, "Do you need a blockchain?," in *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pp. 45–54, IEEE, 2018.

[16] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[17] M. Saad, J. Spaulding, L. Njilla, C. Kamhoua, S. Shetty, D. Nyang, and D. Mohaisen, "Exploring the attack surface of blockchain: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 1977–2008, 2020.

[18] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 3–16, 2016.

[19] "Proof-of-work (PoW)." `https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/`, Aug. 2021.

[20] "Vendor directory hyperledger." `https://www.hyperledger.org/use/vendor-directory`.

[21] "Linux foundation - decentralized innovation, built with trust.." `https://www.linuxfoundation.org`.

[22] M. Graf, R. Küsters, and D. Rausch, "Accountability in a permissioned blockchain: formal analysis of hyperledger fabric," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 236–255, IEEE, 2020.

[23] "Hyperledger fabric whitepaper." `https://www.hyperledger.org/wp-content/uploads/2020/03/hyperledger_fabric_whitepaper.pdf`.

[24] L. Foundation, "Hyperledger fabric docs." `https://hyperledger-fabric.readthedocs.io/en/release-2.5/`, 2024.

[25] S. Maheshwari, "Blockchain basics: Hyperledger fabric IBM developer," 2018.

[26] N. Berendea, H. Mercier, E. Onica, and E. Riviere, "Fair and efficient gossip in hyperledger fabric," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pp. 190–200, IEEE, 2020.

[27] "Leveldb." `https://github.com/syndtr/goleveldb/`.

[28] "The ordering service â hyperledger-fabricdocs main documentation,"

[29] "Apache CouchDB." `http://couchdb.apache.org`.

[30] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm (extended version)." `https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14`, 2013.

[31] "etcd: Distributed reliable key-value store for the most critical data of a distributed system." `https://etcd.io/`.

[32] M. Bishop, *Introduction to computer security, "Key Management, chap.9", 9.5-Digital Signatures page-137.* Addison-Wesley, 2008.

[33] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," *arXiv preprint arXiv:1807.04938*, 2018.

[34] A. Shamir, "Identity-based cryptosystems and signature schemes," in *Workshop on the theory and application of cryptographic techniques*, pp. 47–53, Springer, 1984.

[35] D. Boneh and M. Franklin, "Identity-based encryption from the weil pairing," in *Annual international cryptology conference*, pp. 213–229, Springer, 2001.

[36] N. Z. Aitzhan and D. Svetinovic, "Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 840–852, 2016.

[37] Y. Lindell and A. Nof, "Fast secure multiparty ecdsa with practical distributed key generation and applications to cryptocurrency custody," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1837–1854, 2018.

[38] Y. Wang, "A review of threshold digital signature schemes," 2020.

[39] S. Ergezer, H. Kinkelin, and F. Rezabek, "A survey on threshold signature schemes," *Network*, vol. 49, 2020.

[40] T. F. â. H.-F. M. Documentation, "https://hyperledger-fabric.readthedocs.io/en/release-2.0/txflow.html." `https://hyperledger-fabric.readthedocs.io/en/release-2.0/txflow.html`, 2021.

[41] R. Baldoni, J.-M. Helary, and M. Raynal, "From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pp. 273–282, IEEE, 2000.

[42] "The raft consensus algorithm." `https://raft.github.io`, 2020.

[43] L. Lamport, "The part-time parliament," in *Concurrency: the Works of Leslie Lamport*, pp. 277–317, 2019.

[44] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[45] H. Howard and R. Mortier, "Paxos vs raft: Have we reached consensus on distributed consensus?," in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pp. 1–9, 2020.

[46] O. Alkadi, N. Moustafa, and B. Turnbull, "A review of intrusion detection and blockchain applications in the cloud: approaches, challenges and solutions," *IEEE Access*, vol. 8, pp. 104893–104917, 2020.

[47] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the Works of Leslie Lamport*, pp. 203–226, 2019.

[48] S. De Angelis, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, and V. Sassone, "Pbft vs proof-of-authority: Applying the cap theorem to permissioned blockchain," 2018.

[49] H. Sukhwani, J. M. Martínez, X. Chang, K. S. Trivedi, and A. Rindos, "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, 2017.

[50] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OsDI*, vol. 99, pp. 173–186, 1999.

[51] Y. Wang, "Byzantine fault tolerance in partial synchronous networks," 2020.

[52] W. Jiang, L. Chen, Y. Wang, and S. Qian, "An efficient byzantine fault-tolerant consensus mechanism based on threshold signature," in *2020 International Conference on Internet of Things and Intelligent Applications (ITIA)*, pp. 1–5, IEEE, 2020.

[53] E. Buchman, *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[54] A. Barger, Y. Manevich, H. Meir, and Y. Tock, "A byzantine fault-tolerant consensus library for hyperledger fabric," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 1–9, IEEE, 2021.

[55] github, "In search of an understandable consensus algorithm (extended version)." `https://githubcom/tendermint/tendermint`, 2019.

[56] A. Bessani, J. Sousa, and E. E. Alchieri, "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, 2014.

[57] A. Bessani, J. Sousa, and M. Vukolić, "A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform," in *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, pp. 1–2, 2017.

[58] H. F. Community, "Regarding byzantine fault tolerance in hyperleder fabric.." `https://lists.hyperledger.org/g/fabric/topic/17549966\#3135`, 2018.

[59] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[60] N. V. Database, "Nvd - CVE-2021-30246." `https://nvd.nist.gov/vuln/detail/CVE-2021-30246`, 2021.

[61] T. S. L. Nguyen, G. Jourjon, M. Potop-Butucaru, and K. L. Thai, "Impact of network delays on hyperledger fabric," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 222–227, IEEE, 2019.

[62] M. Saad, J. Spaulding, L. Njilla, C. Kamhoua, S. Shetty, D. Nyang, and A. Mohaisen, "Exploring the attack surface of blockchain: A systematic overview," *arXiv preprint arXiv:1904.03487*, 2019.

[63] R. McMillan, ""the inside story of mt. gox, bitcoinâs $ 460 million disaster."." `https://www.wired.com/2014/03/bitcoin-exchange`.

[64] N. Andola, M. Gogoi, S. Venkatesan, S. Verma, *et al.*, "Vulnerabilities on hyperledger fabric," *Pervasive and Mobile Computing*, vol. 59, p. 101050, 2019.

[65] A. Davenport, S. Shetty, and X. Liang, "Attack surface analysis of permissioned blockchain platforms for smart cities," in *2018 IEEE International Smart Cities Conference (ISC2)*, pp. 1–6, IEEE, 2018.

[66] X. Xian, Y. Zhou, Y. Guo, Z. Yang, and W. Liu, "Improved consensus mechanisms against censorship attacks," in *2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS)*, pp. 718–723, IEEE, 2019.

[67] F. Winzer, B. Herd, and S. Faust, "Temporary censorship attacks in the presence of rational miners," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 357–366, IEEE, 2019.

[68] B. Vitalik, "Automated censorship attack rejection (2017)." `https://silo.tips/download/automated-censorship-attack-rejection-buterin-aug-2017`, 2017.

[69] "Apache kafka," *Developed by LinkedIn, subsequently open-sourced in early*, 2011.

[70] "Hyperledger fabric consensus protocols release-2.5 orderer/consensus package." https://github.com/hyperledger/fabric/tree/release-2.5/orderer/consensus, 2023.

[71] "grpc (grpc remote procedure calls)." https://grpc.io/.

[72] "openblockchain — github.com." https://github.com/openblockchain, 2015.

[73] B. Nguyen, "Hyperledger Fabric - A Brief History — linkedin.com." https://www.linkedin.com/pulse/hyperledger-fabric-brief-history-binh-nguyen/?trackingId=npAL2EodQECJPdouPaw59A%3D%3D, 2017.

[74] P. Thakkar, S. Nathan, and B. Viswanathan, "Performance benchmarking and optimizing hyperledger fabric blockchain platform," in *2018 IEEE 26th international symposium on modeling, analysis, and simulation of computer and telecommunication systems (MASCOTS)*, pp. 264–276, IEEE, 2018.

[75] "The go programming language." https://go.dev/.

[76] "registrar.go source code file." https://github.com/BDLS-bft/fabric/blob/main-bdls/orderer/common/multichannel/registrar.go.

[77] "chainsupport.go source code file." https://github.com/BDLS-bft/fabric/blob/main-bdls/orderer/common/multichannel/chainsupport.go.

[78] "orderer/common/localconfig/config.go source code file." https://github.com/BDLS-bft/fabric/blob/main-bdls/orderer/common/localconfig/config.go.

[79] "Whatâs new in hyperledger fabric v3.0." https://hyperledger-fabric.readthedocs.io/en/latest/whatsnew.html#what-s-new-in-hyperledger-fabric-v3-0.

[80] "Pluggable consensus." https://hyperledger-fabric.readthedocs.io/en/latest/whatis.html#pluggable-consensus.

[81] A. Al Salih and Y. Wang, "Pluggable consensus in hyperledger fabric," in *Proceedings of the 2024 6th Blockchain and Internet of Things Conference*, BIOTC '24, (New York, NY, USA), p. 92 100, Association for Computing Machinery, 2024.

[82] V. Aleksieva, H. Valchanov, and A. Huliyan, "Implementation of smart-contract, based on hyperledger fabric blockchain," in *2020 21st International Symposium on Electrical Apparatus & Technologies (SIELA)*, pp. 1–4, IEEE, 2020.

[83] Y. Wang, "Byzantine fault tolerance in partially synchronous networks," *Cryptology ePrint Archive*, 2019.

[84] Q. Nasir, I. A. Qasse, M. Abu Talib, A. B. Nassif, *et al.*, "Performance analysis of hyperledger fabric platforms," *Security and Communication Networks*, vol. 2018, 2018.

[85] H. Sukhwani, N. Wang, K. S. Trivedi, and A. Rindos, "Performance modeling of hyperledger fabric (permissioned blockchain network)," in *2018 IEEE 17th international symposium on network computing and applications (NCA)*, pp. 1–8, IEEE, 2018.

[86] H. Fabric, "Hyperledger fabric v3.0." `https://hyperledger-fabric.readthedocs.io/en/latest/whatsnew.html`, 2024.

[87] "Hyperledger fabric v3.0." `https://hyperledger-fabric.readthedocs.io/en/latest/whatsnew.html`, 2024.

[88] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus in the lens of blockchain," *arXiv preprint arXiv:1803.05069*, 2018.

[89] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino, "State machine replication in the libra blockchain," *The Libra Assn., Tech. Rep*, vol. 7, 2019.

[90] "Bdls fabric project repository." `https://github.com/BDLS-bft/fabric`.

[91] "Bdls fabric samples / test network repository." `https://github.com/BDLS-bft/fabric-samples`.

[92] "Bdls fabric sdk repository." `https://github.com/BDLS-bft/fabric-sdk-java`.

[93] "Orderer server main.go source code file bdls consensus protocol github organization." `https://github.com/BDLS-bft/fabric/blob/main-bdls/orderer/common/server/main.go`, 2021.

[94] A. Al Salih, "Byzantine Fault Tolerant Consensus For Hyperledger Fabrics," June 2023.

[95] "Using the Fabric test network hyperledger-fabricdocs main documentation — hyperledger-fabric.readthedocs.io." `https://hyperledger-fabric.readthedocs.io/en/latest/test_network.html`, 2014.

[96] "configtxgen hyperledger-fabricdocs main documentation — hyperledger-fabric.readthedocs.io." `https://hyperledger-fabric.readthedocs.io/en/latest/commands/configtxgen.html`, 2014.

[97] "GitHub - BDLS-bft/fabric-samples: Samples for Hyperledger Fabric — github.com." `https://github.com/BDLS-bft/fabric-samples`. [Accessed 02-Jun-2023].

[98] "osnadmin channel." https://hyperledger-fabric.readthedocs.io/en/latest/commands/osnadminchannel.html.

[99] "fabric-samples/test-network-nano-bash at main hyperledger/fabric-samples-github.com." https://github.com/hyperledger/fabric-samples/tree/main/test-network-nano-bash. [2023].

[100] "Deploy the ordering service hyperledger-fabricdocs main documentation — hyperledger-fabric.readthedocs.io." https://hyperledger-fabric.readthedocs.io/en/release-2.5/deployorderer/ordererdeploy.html, 2023.

[101] H. Honar Pajooh, M. Rashid, F. Alam, and S. Demidenko, "Hyperledger fabric blockchain for securing the edge internet of things," *Sensors*, vol. 21, no. 2, p. 359, 2021.

[102] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[103] C. Cachin and M. Vukolić, "Blockchain consensus protocols in the wild," *arXiv preprint arXiv:1707.01873*, 2017.

[104] A. Baliga, "Understanding blockchain consensus models," *Persistent*, vol. 4, no. 1, p. 14, 2017.

[105] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, "On blockchain and its integration with iot. challenges and opportunities," *Future generation computer systems*, vol. 88, pp. 173–190, 2018.

[106] I. Butun, P. Österberg, and H. Song, "Security of the internet of things: Vulnerabilities, attacks, and countermeasures," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, pp. 616–644, 2019.

[107] R. Han, G. Shapiro, V. Gramoli, and X. Xu, "On the performance of distributed ledgers for internet of things," *Internet of Things*, vol. 10, p. 100087, 2020.

[108] A. Al Salih and Y. Wang, "Securing the connected world: Fast and byzantine fault tolerant protocols for iot, edge, and cloud systems," in *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, pp. 34–41, 2024.

[109] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn, "Corda: an introduction," *R3 CEV, August*, vol. 1, no. 15, p. 14, 2016.

[110] B. Farahani, F. Firouzi, and M. Luecking, "The convergence of iot and distributed ledger technologies (dlt): Opportunities, challenges, and solutions," *Journal of Network and Computer Applications*, vol. 177, p. 102936, 2021.

[111] L. Lao, X. Dai, B. Xiao, and S. Guo, "G-pbft: a location-based and scalable consensus protocol for iot-blockchain applications," in *2020 IEEE international parallel and distributed processing symposium (IPDPS)*, pp. 664–673, IEEE, 2020.

[112] C. Stathakopoulou, D. Tudor, M. Pavlovic, and M. Vukolić, "Mir-bft: Scalable and robust bft for decentralized networks," *Journal of Systems Research*, vol. 2, no. 1, 2022.

[113] J. Mišić, V. B. Mišić, X. Chang, and H. Qushtom, "Adapting pbft for use with blockchain-enabled iot systems," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 1, pp. 33–48, 2020.

[114] J. Mišić, V. B. Mišić, X. Chang, and H. Qushtom, "Multiple entry point pbft for iot systems," in *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pp. 1–6, IEEE, 2020.

[115] "Benchmarking hyperledger fabric 2.5 performance." `https://www.hyperledger.org/blog/2023/02/16/benchmarking-hyperledger-fabric-2-5-performance`.

[116] Z. Shi, H. Zhou, Y. Hu, S. Jayachander, C. de Laat, and Z. Zhao, "Operating permissioned blockchain in clouds: A performance study of hyperledger sawtooth," in *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 50–57, IEEE, 2019.

[117] M. Shuaib, N. H. Hassan, S. Usman, S. Alam, N. A. A. Bakar, and N. Maarop, "Performance evaluation of dlt systems based on hyper ledger fabric," in *2022 4th International Conference on Smart Sensors and Application (ICSSA)*, pp. 70–75, IEEE, 2022.

[118] W. Choi and J. W.-K. Hong, "Performance evaluation of ethereum private and testnet networks using hyperledger caliper," in *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pp. 325–329, IEEE, 2021.

[119] A. A. Salih and Y. Wang, "Bdls as a blockchain finality gadget: Improving byzantine fault tolerance in hyperledger fabric," *IEEE Access*, pp. 1–1, 2024.

APPENDIX A: QUADRATIC FIT COMPARISON GRAPHS

A.1 Phase 1: New block Propose

Algorithm 1 Phase 1: New block Propose

Data: B'
Result: $P_i \leftarrow (\langle h, r\rangle_j, \langle h, r, B'_j)_j\rangle$
$P_j \leftarrow B'$
$P_i \leftarrow B'$
where $B'_j \in \text{BLOCK}_j$
while $r' < r$ **do**
  if $(\langle h, r\rangle_j, \langle h, r, B'_j)_j\rangle$ *signed* **then**
    $P_i \leftarrow (\langle h, r\rangle_j, \langle h, r, B'_j)_j\rangle$
    where $B'_j \in \text{BLOCK}_j$
    round change $\leftarrow (h, r)_j$
  end
  ; /* P_j will not accept messages except a ‘‘decide’’ */
  if *messages is “decide”* **then**
    $Pj \leftarrow$ *messages*
  end
end

Algorithm 2 Phase 2: Lock

Data: B'
Result: $\langle lock, h, r, B', proof\rangle_i or \langle select, h, r, B'', proof\rangle_i$
$Where B'_j \in \text{BLOCK}_j$
while $B' \neq NULL.$ **do**
  if $P_i \leftarrow 2t+1$ *signed* **then**
    $P_i \leftarrow \langle lock, h, r, B', proof\rangle_i \xrightarrow{\text{broadcast}}$ message to all participants P_j
    "proof" $\in \langle h, r, B'\rangle$
  end
  else
    Where B'' is the candidate block
    $B'' = \max\{B : B \in \text{BLOCK}_i\}$
    P_i**Broadcast** $\langle select, h, r, B'', proof\rangle_i$
    "proof" $\in \langle h, r\rangle$
  end
end

A.3 Phase 3: commit

Algorithm 3 Phase 3: commit

Data: $\langle lock, h, r, B', proof \rangle_i$ or $\langle select, h, r, B'', proof \rangle_i$
Result: $\langle commit, h, r, B' \rangle_i$
$P_i, P_j \leftarrow 2t + 1$ signed messages
$P_i \leftarrow B'$
Where $B'_j \in \text{BLOCK}_j$
while $2t + 1$ *signed messages.* **do**
  if $P_i, P_j \leftarrow \langle lock, h, r, B', proof \rangle_i$ **then**
    1. locks on B' from the previous round are released,
       No other potential locks are released
    2. locks B' the candidate block
    3. /*All nodes send commit message to the leader*/
    $P_i \leftarrow \langle commit, h, r, B' \rangle_j$
  end
  else
    if $P_i, P_j \leftarrow \langle select, h, r, B'', proof \rangle_i$ **then**
      Add B'' to BLOCK_j
    end
  end
end

Algorithm 4 Phase 4: decide

Data: $\langle commit, h, r, B' \rangle_j$
Result: $\langle decide, h, r, B', proof \rangle_i$
$P_i \leftarrow 2t + 1$
$P_i \leftarrow B'$
$WhereB'_j \in \text{BLOCK}_j$
while P_i *decides on the value* B' **do**
  if $P_i \leftarrow 2t + 1$ *commit messages* **then**
    $P_i \leftarrow \langle decide, h, r, B', proof \rangle_i$
    $P_i \xrightarrow{\text{broadcast}} \langle decide, h, r, B', proof \rangle_i$ to all participants P_j, P_i
    "proof" $\in \langle h, r, B' \rangle$
  end
end