Understanding Scalability Issues in Sharded Blockchains

by

Anh Duong Nguyen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Some of the source code, text and figures in Chapter 3 are borrowed from our joint poster publication co-authored by myself, my supervisor, Dr. Golab, and a PhD student, Mr. Chunyu Mao [51]. The system architecture discussed in this thesis was jointly designed by me, Dr. Golab and Mr. Mao, and the software prototypes were co-developed with Mr. Mao. I implemented the additional components described in Chapter 4, developed the testing scripts and carried out the experiments in Sections 3.4 and 4.3.

Abstract

Since the release of Bitcoin in 2008, cryptocurrencies have attracted attention from academia, government, and enterprises. Blockchain, the backbone ledger in many cryptocurrencies, has shown its potential to be a data structure carrying information over the network securely without the need for a centralized trust party. In this thesis, I delve into the consensus protocols used in permissioned blockchains and analyze the sharding technique that aims to improve the scalability in blockchain systems. I discuss a permissioned sharded blockchain that I use to examine different methods to interleave blocks, referred to as strong temporal coupling and weak temporal coupling. I provide empirical experiments to show the roles of lightweight nodes in solving the scalability issues in sharded blockchain systems. The results suggest that the weak temporal coupling method performs worse than the strong temporal coupling method and is more susceptible to an increase in network latency. The results also show the importance of separating the roles of nodes and adding lightweight nodes to improve the performance and scalability of sharded blockchain systems.

Acknowledgements

I would like to thank my supervisor, Dr. Wojciech Golab, for all of the support and motivation throughout my study at the University of Waterloo. This thesis would not be possible without his keen insights and advice. I would also like to thank Dr. Paul Ward and Dr. Mahesh V. Tripunitara for providing their valuable feedback on my thesis. In addition, I would like to thank Mr. Chunyu Mao for his collaboration in our ICBC poster paper [51]. I want to thank Ripple and the Faculty of Engineering for sponsoring this research. Last but not least, I also want to express my gratitude to our former Dean of Engineering, Dr. Pearl Sullivan, for her enormous contribution to the Faculty of Engineering and the University of Waterloo.

Dedication

This is dedicated to my family and my significant other. Thank you for your constant love and support.

Table of Contents

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Permissionless vs. Permissioned Blockchain 2

1.2 Scalability Issues of Blockchain 3

1.3 Sharding in Blockchain 3

1.4 Contributions and Organization 4

2 Literature Review 5

2.1 The Consensus Problem 5

2.1.1 Paxos and Egalitarian Paxos 7

2.2 Blockchain Protocols 10

2.2.1 Permissionless Blockchain and BFT Protocols 11

2.2.2 Permissioned Blockchain and BFT Protocols 15

2.3 Sharded Blockchain Protocols 17

2.3.1 Elastico 18

2.3.2 OmniLedger 19

2.3.3 Rapid Chain 20

2.3.4 Learnings 21

3 Interleaving Blocks in a Sharded Permissioned Blockchain **23**
3.1 Methods of Interleaving . . . 24
3.2 System Architecture . . . 25
3.3 Implementation . . . 28
3.3.1 Protocol Buffers and gRPC . . . 28
3.3.2 Concurrency in Go . . . 30
3.3.3 Functionalities in Front-End Servers . . . 30
3.3.4 Pseudo-Code Description . . . 31
3.4 Evaluation . . . 34
3.4.1 Simulated Environment . . . 34
3.4.2 On AWS EC2 . . . 39
3.4.3 Summary . . . 41

4 Lightweight Front-End Servers **42**
4.1 Motivation . . . 42
4.1.1 Simplified Payment Verification . . . 42
4.1.2 Bloom Filter . . . 43
4.2 Implementation . . . 45
4.3 Evaluation . . . 49
4.3.1 Adding Lightweight Front-End Servers to the System . . . 50
4.3.2 Varying Bloom Filter Sizes . . . 53
4.3.3 Ratio of Full Front-End Servers in a System . . . 55

5 Conclusion and Future Work **57**

References **59**

APPENDICES **67**

A FLP Impossibility **68**

List of Figures

2.1 Bitcoin block structure 12
2.2 Bitcoin transaction structure 12
2.3 A typical sharding mechanism in blockchain. 18

3.1 Methods of interleaving 24
3.2 System design for comparing methods of interleaving. 26
3.3 Block structure 27
3.4 Transaction structure 27
3.5 Common functions of strong and weak coupling FE servers 34
3.6 Strong-coupling front-end servers pseudo-code 35
3.7 Weak-coupling front-end servers pseudo-code 36
3.8 Performance of strong and weak temporal coupling methods in simulated environment with no simulated network latency 37
3.9 Performance of strong and weak temporal coupling methods in simulated environment with a 10ms round trip simulated latency 38
3.10 Performance of strong and weak temporal coupling methods on AWS EC2 40

4.1 A Bloom filter with $m = 8$, $n = 2$ and $k = 2$. 44
4.2 A Bloom filter with a collision vector. 44
4.3 Adding lightweight front-end servers to the system design 46
4.4 Common functions in full and lightweight front-end servers 47
4.5 Strong-coupling full front-end servers pseudo-code 48

4.6 Weak-coupling full front-end servers pseudo-code 49
4.7 Lightweight front-end servers pseudo-code 50
4.8 Four ways to distribute blocks from back-end servers to front-end servers . 52
4.9 Compare the performances of strong and weak coupling methods, with and without using lightweight front-end servers. 53
4.10 Performance with and without using Bloom filters to check double-spending within each block. 54
4.11 Peak throughputs with different number of full front-end servers 56

List of Tables

3.1 Round-trip latency across AWS regions 39

Chapter 1

Introduction

Over the past decade, blockchain technology has attracted attention from both academia and industry. After its introduction in the Bitcoin white paper [61] in 2008, blockchain has been serving as the backbone of Bitcoin and other succeeding cryptocurrencies, such as Ethereum [76], Litecoin [26], etc. Blockchain technology has emerged with a promise to securely send and share a large amount of data in a peer-to-peer manner without the need for traditional centralized authority. Besides the cryptocurrency markets, governments, academia, and private companies are exploring other areas where blockchain technology can be applied [53, 81]. Some of the potential applications include smart contracts (insurance, legally binding agreements, etc.), data confidentiality in healthcare, and supply-chain management [13, 54, 66]. Contrary to the traditional financial system, each participant in a blockchain actively contributes to the correctness of the system and may not trust each other. Blockchain systems can provide distributed, highly fault-tolerant systems while the participants can remain pseudo-anonymous. The content of a blockchain is shared publicly and resistant to modification. There is no central authority that can corrupt a blockchain or tamper with its content. Any modification requires the collaboration and agreement of a majority of participants in the network. Diminishing the role of a central authority allows blockchain systems to enhance their security and eliminate a single point of failure.

Despite its promising applications, blockchain has been facing challenges to compete with existing centralized services. At the heart of those challenges is the widely discussed issue of scalability [12, 34]. Several blockchain systems have used the sharding technique to improve scalability and reduce space overhead [24, 37, 50, 79]. In sharded blockchain systems, the computational workload and blocks are partitioned into multiple shards, where each shard maintains an independent blockchain and communicates with each other to handle cross-shard transactions.

In this thesis, I compare different methods to interleave blocks in sharded blockchains and propose a novel architecture that improves the scalability. Concretely, I first analyze several existing sharded blockchains, their consensus protocols, and their mechanisms to handle intra-shard and cross-shard transactions. Second, I present the design and implementation of a novel permissioned sharded blockchain. Using this blockchain, I compare two methods to interleave blocks from different shards, referred to as strong and weak temporal coupling. I show that under less-than-ideal network conditions, the strong temporal coupling method improves the scalability of the blockchain system. Finally, I propose, implement, and show that using lightweight nodes coupled with a Bloom filter to check double-spending can substantially improve the scalability of the blockchain system.

1.1 Permissionless vs. Permissioned Blockchain

There are two categories of blockchain systems: permissionless and permissioned. In a permissionless blockchain, any node can join and leave the network without permission. To join a permissionless blockchain network such as Bitcoin [61], a node only needs a pair of public and private keys to sign their messages and create an address to receive coins. This allows any user to join anonymously and makes it possible for an adversary to create multiple identities to increase the weight of their vote in the system. This type of attack is referred to as a Sybil attack [21]. Permissionless blockchains have proposed different types of consensus to solve this problem, such as Proof-of-Work [60] and Proof-of-Stake [27], where nodes cast their votes using their computational power or the number of stakes. These types of consensus protocols do not limit the number of participants and are public and decentralized, but inefficient in scaling [80].

A permissioned blockchain controls the group of participants, as opposed to permissionless blockchains [29, 64]. That can be suitable for applications where participants need permission to join and need to share their identities, although they may not need to have mutual trust. By granting permissions to a specific group of participants, permissioned blockchains can generally solve more complex problems with higher throughput and lower latency. They can be particularly useful in permissioned enterprises where they need permission to access information in blockchains. The applications of permissioned blockchains have attracted attention in private enterprises. For example, the Hyperledger Fabric [3] is a permissioned blockchain protocol that can serve as a foundation of blockchain systems in private companies. Each node in the Hyperledger Fabric must have a known identity and requires permission to access information in the network. In some sectors, such as finance and healthcare, it is essential to share and manage permissions to access data while

maintaining privacy. Such applications can benefit from using private blocks for regulating data storage, permission, and distribution.

1.2 Scalability Issues of Blockchain

Scalability is a characteristic of a system that can improve performance by adding more resources. In blockchain, scalability is a term related to several quantitative metrics. For example, in [18], scaling a blockchain system refers to any improvement in the perspective of throughput, latency, bootstrap time, or cost per transaction. A widely-discussed metric in evaluating a blockchain system is its peak throughput, which is the maximum number of requests that a blockchain system can process per unit of time. In blockchain protocols similar to Bitcoin, each request is a transaction. The throughput of a blockchain system is generally affected by the frequency of blocks and their sizes. For example, in Bitcoin, those parameters are fixed. One block can have a maximum of 1MB transactions, and one block is mined approximately every 10 minutes [60]. A Bitcoin transaction with one input and two outputs takes about 250 bytes, which means Bitcoin can have a maximum of 4200 transactions every 10 minutes. A second metric to measure the performance of a blockchain system is latency. A transaction is processed when it is confirmed in a valid block, and the latency indicates the time it takes to confirm a transaction. A shorter latency means a shorter waiting time, and hence extends the ability of blockchains to serve in various applications.

Currently, Bitcoin has an average throughput of only 7 transactions per second [74], and the chain adds around 50GB of data every year. Meanwhile, on average, Visa processes 2000 transactions per second. Besides, it takes Bitcoin around 10 minutes to create a block and an hour to confirm the validity of the transactions [61]. Improving the scalability in blockchains is crucial to exploit its full potential, making it possible for blockchain-based applications to achieve performance comparable to existing centralized technologies. This has raised new challenges and questions to the future designs of blockchain so that it can handle a higher workload while maintaining its decentralized nature.

1.3 Sharding in Blockchain

Sharding is a well-known approach used in the traditional database to spread the load horizontally [5], widely used in distributed databases such as Spanner [15]. It allows the

distribution of data on a wide range of data centers, and this approach can improve performance when a query requires only a subset of shards. Several works have been using the sharding technique to solve the scalability problem of blockchain [24, 37, 50, 79]. In blockchains, sharding is a way to partition consensus workload and transactions to multiple nodes. The blocks and the computational workload are separated into smaller chunks and distributed to different nodes and not all nodes need to store the entire chain or do all of the computational work (such as verifying cryptographic signatures). The nodes are partitioned into shards, where each shard can act independently as a secure, distributed ledger. This technique enables the parallelization of processing, storage, and computing. Hence it can help to achieve a scale-out blockchain system by reducing the overhead of communication, computation, and data storage. However, sharding has also brought some problems in designing the intra-shard and cross-shard consensus. This issue will be later discussed in detail in Section 2.3.

1.4 Contributions and Organization

The thesis is organized as follows. In Chapter 2, I discuss several consensus protocols in blockchains, analyze the sharding technique, and how it is used in sharded blockchain systems. I present and compare two popular approaches to handle cross-shard transactions, which are maintaining a root chain or building a full-mesh connection among shards. In Chapter 3, I present a novel permissioned sharded blockchain and investigate two methods to interleave blocks from different shards to form a root chain and refer to them as strong and weak temporal coupling methods. I show that the weak temporal coupling method is more susceptible to the increase in network latency and using this method reduces the throughput of the blockchain system. In Chapter 4, I propose a novel way to use Bloom filters to help check double-spending transactions and use lightweight servers to improve the scalability. I also show how the size of the Bloom filter impacts the overall throughput and latency of the system. I provide empirical results to show the importance of assigning roles to nodes and choosing the constants (such as the number of validators and committee sizes) in sharded blockchains. In Chapter 5, I summarize the overall results and propose future works.

Chapter 2

Literature Review

Over the past decade, blockchain has become an emerging technology that allows secure information sharing over peer-to-peer networks. It has shown enormous potential to perform digital transactions without a centralized authority. An example of such a system is Bitcoin - the first cryptocurrency implemented using a blockchain. The novelty in blockchain design was derived from research results in distributed computing, cryptography, and game theory over several decades. The core of a blockchain system is a secure, distributed ledger with a consensus protocol that guarantees consistency despite the presence of malicious participants and attacks. Therefore, to understand the scalability problem in blockchain, it is important to study different consensus mechanisms in distributed computing in depth.

2.1 The Consensus Problem

The consensus problem is a classical problem in distributed computing. Informally speaking, a multi-processes system reaches consensus when a single value or action among the proposed values is chosen, despite some processes being faulty. A process is faulty if it does not fulfill its role in the protocol and is unreliable in some ways. For example, it may fail to respond to messages, send erroneous messages, or give conflicting information to different peers. A secure consensus protocol must tolerate such faults under some assumptions of the system model. These assumptions must indicate what is the minimum required number of non-faulty nodes, how a process can fail (e.g., crash-failure, Byzantine failure, etc.), and the type of network under which the protocol is operating (e.g., synchronous or asynchronous message-passing mechanisms; reliable or unreliable communication links; unicast or multicast communication, etc.). A crash failure happens when a process works

correctly until it abruptly halts and fails to respond to or send messages. A consensus protocol that can tolerate crash failures must satisfy 3 properties:

- **Termination:** Eventually, every non-faulty process must decide on some value.
- **Agreement:** No two processes can decide on different values.
- **Validity:** If a value v is decided, it must be proposed by some process.

A Byzantine failure happens when a system fails arbitrarily due to a Byzantine fault. In the Byzantine failure model, a faulty process can exhibit any behavior. For example, it can change its state arbitrarily, delay sending messages, send no message or erroneous messages to peers, or send different messages to different peers. The Byzantine fault is named after the "Byzantine Generals Problem," first described in [42]. The problem describes a scenario when multiple generals are attacking a common enemy and at least 3 of them must agree on a time to attack. All generals have to communicate with and send messages to each other while one or more generals can lie about their choice. The Byzantine failure is considered the most general and difficult class of failures. The 3 properties for a consensus protocol that can tolerate Byzantine failures are:

- **Termination:** Eventually, every non-faulty process must decide on some value.
- **Agreement:** No two correct processes can decide on different values.
- **Validity:** If a value v is proposed by every non-faulty process, then all of them must decide on v.

A consensus protocol that provides *safety* must satisfy two properties Agreement and Validity, and guarantees *liveness* when it satisfies the Termination property.

In a system model, most consensus protocols are assumed to work under reliable communication links. Processes use message-passing mechanisms to coordinate and communicate with each other. There are two types of message-passing environment: synchronous and asynchronous [75]. Synchronous models presume that all processes must receive a message, complete some work, and reply within a bound time. If other peers fail to receive some responses from a process P within the bound time, they presume that process P has crashed. In an asynchronous setting, there is no upper bound time that processes may take to receive, process, and reply to an incoming message. It is impossible to detect if a process has crashed, or it is just taking a long time to finish.

As proved in [63], if faulty processes are allowed to lie, a synchronous system that has reliable communication links needs at least $3n+1$ processes to tolerate n failures. In [23], the Fischer-Lynch-Paterson (FLP) proof shows that in an asynchronous setting where messages can be delayed, there is no deterministic consensus protocol that is guaranteed to terminate (i.e., achieves liveness) if there is at least one crash (see Appendix A). There are two ways to circumvent the FLP result: ① using randomization to achieve liveness and safety with an arbitrarily high probability, and ② using some synchrony assumptions. For example, Paxos [40] works in an asynchronous setting and guarantees safety but fails to guarantee liveness. PBFT [10] can achieve safety under asynchrony, but requires synchrony to achieve liveness. In Bitcoin, the average time between the creation of two consecutive blocks is roughly 10 minutes. The environment is loosely synchronous under the assumption that after 10 minutes, a block reaches every process despite any prolonged propagation time. In Bitcoin, safety is achieved with a high probability. If the hashing power of an adversary in Bitcoin is 10% and there are 5 blocks linked after a block B, the probability for the adversary to create a longer chain and invalidate block B is 0.1% [61]. The current total hashing power of Bitcoin is over 100 quintillion hashes per second [7]. That means to have 0.1% chance of reversing a block with 5 subsequent blocks, an adversary must be able to perform over 10 quintillion hashes per second. Thus, in practice, a Bitcoin transaction requires at least 5 further attached blocks to be considered valid. HoneyBadger [56] takes a random approach and uses an algorithm that randomly bundles transactions and uses threshold signatures [8] to achieve liveness and safety with a high probability of success.

2.1.1 Paxos and Egalitarian Paxos

Paxos is a well-known algorithm in distributed computing for solving consensus in a network where some nodes are unreliable [40, 42]. Egalitarian Paxos [58] is an efficient, leaderless variant of Paxos. It better tolerates slow replicas by decoupling them from the fastest and balancing load among replicas both in the local and wide area. Its open-source implementation is our consensus protocol for the prototypes in Chapters 3 and 4.

Paxos

Paxos [40] solves the consensus problem in an asynchronous setting under the assumption that there is no Byzantine fault. It is a protocol that helps a group of processes agree on a proposed value and other processes learn that value. For safety, only a single proposed value can be chosen, and a process cannot learn a value until it has been decided. We assume that all processes have access to stable storage and remember all messages they

have received and promised. There are 3 types of agents in Paxos: *proposers*, *acceptors*, and *learners*. Paxos processes can play multiple roles and even all of them. Proposers and acceptors actively participate in the consensus protocol to decide the output value. Each proposer sends a value v to a group of acceptors, and if a majority of acceptors accept the value, it is accepted, and then learners can learn the accepted value. Before beginning the protocol, all processes must know how many processes are there and how many acceptors makes a majority, because any two majorities must overlap at least one process. There are two phases in the algorithm:

1. Phase 1:

- A proposer p sends a PREPARE IDp message to at least a majority of acceptors. The id IDp must be unique among different proposers and PREPARE messages sent by the same proposer. For example, it can be the timestamp in nanoseconds concatenated with the proposer identity. Note that the id IDp is not related to the value v that the proposer p wants to propose. If the PREPARE IDp message is timed out, the proposer retries with a higher id.

- When an acceptor receives a message PREPARE IDp, it will ignore this message if it has accepted an id higher than IDp. Otherwise, it accepts this message and promises to ignore all messages with an id lower than IDp. It replies to the proposer with a PROMISE IDp message. If the acceptor has previously accepted a different message with an id IDq and a value v_q, it also sends that id and value to the proposer.

2. Phase 2:

- Once a proposer receives a majority of PROMISE IDp messages from acceptors, it sends a message ACCEPT-REQUEST IDp, VALUE v_p to at least a majority of acceptors. The value v_p is the value v_q from the message with the highest id IDq that the proposer has received from the acceptors. If no such message exists, the proposer p can assign its own value to v_p.

- When an acceptor receives an ACCEPT-REQUEST IDp, VALUE v_p message, it ignores the message if it has promised to do so. Otherwise, it accepts this value and sends a message ACCEPT IDp, VALUE v_p to all the processes, including learners.

A proposer or learner knows that the consensus is reached on a value v_p if it receives the ACCEPT messages from a majority of acceptors. Paxos can tolerate up to f crash failures with $2f + 1$ processes because a majority of acceptors would be $f + 1$. Paxos guarantees

safety properties. However, it does not guarantee liveness. Livelock occurs when a proposer finishes phase 1 right before another proposer enters phase 2. This happens when two or more proposers think that they are the leader of the current run. The result is supported by the FLP impossibility proof [23], which suggests that in a fully asynchronous message-passing distributed system, if at least one process may fail, it is impossible to have a deterministic algorithm that can achieve both safety and liveness.

Egalitarian Paxos

In a Paxos system where there is a stable leader, we can omit phase 1. Many variants of Paxos take this approach [41, 43]. An example is Multi-Paxos [11], where a leader is chosen once, and only it can propose values and broadcast messages to all acceptors and learners. If this leader fails, the protocol requires another round of consensus to elect a new leader. However, Egalitarian Paxos (EPaxos) [58] is a variant where there is no designated leader, and hence it can avoid performance bottleneck and a single point of failure in the system. Clients can send requests to any replica, usually the closest one to reduce network overhead. When there are at most f faulty processes and $f + \lfloor \frac{f+1}{2} \rfloor$ of replicas agree on a value, the system commits within a single round of network communication. The number $f + \lfloor \frac{f+1}{2} \rfloor$ is called a fast-path quorum and for three and five replicas, this number is optimal (two and three replicas respectively).

EPaxos is evaluated in the context of a key-value store. Their read operations are commutative, and hence two different replicas can have different orders of *read* operations but remain in the same state. Two operations interfere if they operate on the same value and one of them is a write. The key idea in EPaxos is to find dependencies among concurrent operations across different replicas and in each replica. EPaxos runs in 3 phases:

1. Phase 1: Establish ordering constraints

   - When a replica L receives a command γ from a client, it becomes a command leader. It prepares a dependency list *dep* of all instances (i.e., command slots) whose commands interfere with γ and a sequence number *seq*, which is greater than those of all commands in *dep*. Then, it sends a message PREACCEPT γ, dep, seq to all replicas.

   - When a replica R receives a PREACCEPT message, it learns the given *dep* and *seq* and records command γ in its command log. It replies with its updated *dep* and *seq*.

   - If L receives replies from a fast-path quorum of replicas and all replies are the same, it will move to the *commit* phase (phase 3). Otherwise, it goes to the Paxos-Accept phase (phase 2), which means some replies are different from the others.

2. Phase 2: Paxos-Accept phase

   - L generates the unions of all *dep* variables and chooses the highest *seq* number. After updating the attributes in its internal state, it tells all other replicas to accept the updated attributes by sending a message ACCEPT dep', seq'.

   - If L hears from a majority of replicas, it moves to the commit phase.

3. Phase 3: Commit phase

   - L sends the commit messages to other replicas.

   - All replicas log the command as committed.

   - L notifies the commit to the client.

After running the above protocol, every non-failing replica can generate a dependency graph of commands and go to the execution phase. Then, each replica finds strongly connected components and sorts them topologically. In each strongly connected component, it sorts all commands using their *seq* and executes the commands in the order.

2.2 Blockchain Protocols

Blockchain is a peer-to-peer ledger containing a growing list of blocks, linked using cryptography. It was introduced by Satoshi Nakamoto [61] to serve as a ledger for the cryptocurrency Bitcoin, making Bitcoin the first cryptocurrency that solves the double-spending problem. In digital cryptocurrency, the double-spending problem occurs when the holder of a digital coin sends it to more than one recipient. Blockchain is now used by most cryptocurrencies and its applications are extended to smart contracts and banking systems. Besides cryptocurrencies and finance-related services, blockchain has the potential to be used in other areas such as identity management (online authentication, privacy-preserving identity, ownership rights), and data security [69].

Blockchain is decentralized and distributed across many computers in the network, so no single one has full control. Blockchain is resistant to data modification and is a suitable option in a distributed network where nodes do not trust each other and every node can have a copy of the data. State-machine replication is a traditional approach to fault tolerance [52]. The data and service are replicated among multiple nodes so that the system continues to work despite the failure of one or more nodes. A blockchain is

considered secure if no node can alter the data in a given block without changing the data in all of its subsequent blocks. By design, this modification is very difficult to make on a blockchain, and the details will be explored later in this chapter. There is no trusted third party, and authentication is achieved by collaboration between participants. No participant is trusted more than another, centralized points of vulnerability are eliminated in blockchain, and so it is considered secure by design.

Blocks are added to a blockchain, one after another, by any node in the network. The first block in a chain is called the genesis block, which generally contains some hardcoded information. Some blockchain systems allow forking, which happens when the chain of blocks diverges into more than one path. In this case, all nodes agree that the longest chain is the only valid one. If a node is working on a chain and later learns that another chain is longer, it immediately switches to the longest chain. If there is more than one longest chain, each node considers the first chain it receives as the longest one. To ensure a global agreement on the state of the blocks, many blockchain systems depend on an underlying consensus mechanism which is Byzantine fault-tolerant (BFT).

2.2.1 Permissionless Blockchain and BFT Protocols

In a permissionless blockchain, anyone can join as a user to send transactions or act as a node that participates in the consensus protocol. Malicious actors can have multiple fake identities to cast faulty votes and modify the content of blocks. This type of attack is referred to as a Sybil attack [21]. Permissionless BFT protocols have been using various techniques such as Proof-Of-Work [61] and Proof-Of-Stake [27] to overcome that problem.

Bitcoin

Blockchain was first introduced by Nakamoto in 2008 in the white paper of Bitcoin [61]. The blockchain in Bitcoin stores the exchange of digital coins from one user to another. Each block has a constant magic number of `0xD9B4BEF9`, a blocksize which is the number of bytes in the block, a block header, and a list of valid transactions represented by a Merkle-tree. Each transaction is a data structure that represents a transfer of digital coins between two or more users. It contains the list of inputs, outputs, and scripts that specify the transaction's source, destination, and the condition for the transfer to happen. The header of a block contains a cryptographic hash of its parent block, Merkle-tree root hash, timestamp, nBits (difficulty target), nonce, and block version [80] (see Figure 2.1). The block version field helps nodes keep track of changes and updates throughout the protocol.

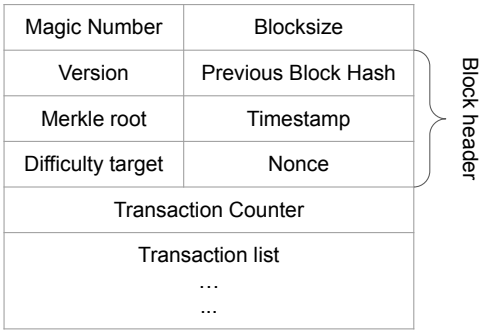


Figure 2.1: Bitcoin block structure

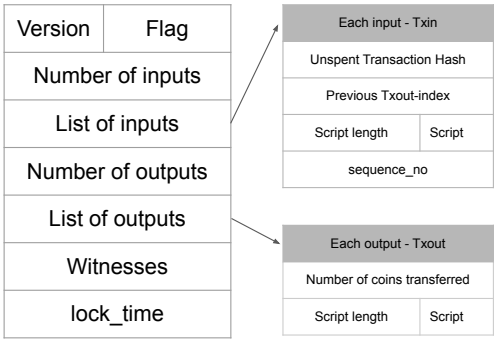


Figure 2.2: Bitcoin transaction structure

The timestamp field is an estimate of the block creation time. A Merkle-tree is a tree of hashes, and each leaf in the tree stores the hash of a transaction. Only the root of the tree is included in the block header. This data structure allows efficient query operations: checking whether a transaction is in the block takes logarithmic time. The previous block hash is a hash of the most recently added block of the blockchain, calculated by applying the SHA-256 cryptographic function twice on the previous block-header. With this design, modifying any transaction in a block will probabilistically change its hash. As a consequence, that will change the previous block hash fields in every subsequent block. This modification is hard to make, and the probability for an adversary to alter a transaction is very low. As shown in [61], if the hashing power of an adversary in Bitcoin is 10% and there are 5 blocks linked after a block B, the probability for the adversary to create a longer chain and invalidate block B is 0.1%. Hence, in practice, the transactions in a block are considered valid and completed if the block has at least 5 subsequent blocks appended after it.

To participate in the network, each user must own a Bitcoin wallet, which contains one private key and one public key to verify their identity. Each wallet has an address to receive transactions from other users, and the address is a hashed version of the public key. The private key is kept secret and used to encrypt data sent to other users. There are typically two phases involved in the digital signature. The first phase is signing: the sender encrypts the data they want to send using their private key. The second phase is verification: another node ensures that it was the sender that encrypted the data. Bitcoin uses the elliptic curve digital signature algorithm (ECDSA) [31] for signing and verifying digital signatures.

Unlike in the traditional banking system, there is no concept of account in the Bitcoin blockchain. To check one's balance, we go through every block and sum up the number of coins that are sent to a Bitcoin wallet and not yet spent. A transaction is considered

spent if its hash is an input of another transaction in a subsequent block. To send a transaction to a receiver, a sender creates a transaction and defines the list of inputs and outputs. The inputs are represented on a list of unspent transactions owned by the sender, and the outputs include a list of addresses of the recipients and the number of transferred coins. After the sender encrypts a transaction, it sends the transaction to a mem pool. Miners compete with each other to get the right to add a new block to the blockchain. The successful miner obtains a batch of transactions from the mem pool, forms a block, and broadcasts it to other nodes in the network. Each node acts independently to decide whether the new block is valid. After 5 subsequent blocks, the transaction is considered valid, and the receiver can spend the coins by including the transaction hash in the input list of another transaction.

Bitcoin solved the BFT problem using Proof-Of-Work, where a participant's vote is not based on their number of identities but their computational power. Any Proof-Of-Work scheme involves a puzzle that is difficult to solve but trivial to verify the answer. For each Bitcoin block that a miner mines, it gets a certain amount of Bitcoin as a reward. In Bitcoin, the puzzle for miners is to calculate the hash of the block and adjust the nonce, which is a 32-bit field in the block header. After appending the nonce to the hashed contents of the block and then rehashing the result, we must get a number that is smaller or equal to the target value in the most recent block appended to the chain. Bitcoin uses the SHA-256 hash function, and the strategy to find a correct nonce is to try different nonces until we find a correct one. That is a random process that requires computational effort, so any adversary that wishes to alter a transaction content must invest in computational power. This is referred to as the 51% attack [9], where one or a group of miners own more than 50% of the total computational power of the network. This group can disrupt the Bitcoin payment system, prevent new transactions from gaining confirmation, reverse transactions in previously mined blocks, and double-spend coins.

Algorand

Algorand [27] is a high-performance permissionless blockchain system that uses a new Byzantine Agreement (BA*) protocol [27]. All users in the Algorand network use BA* to reach consensus on the next valid block to append to the blockchain. BA* uses verifiable random functions [55] (VRFs) to randomly and non-interactively select a subset of participants (a committee) to decide the outcome in each step. Algorand offers four main features:

- Weighted users (i.e., Proof-of-Weight): Sybil attacks are easy to launch in a system

that cheaply generates identities. To avoid them, Algorand weights each user by the number of coins in their account. BA* guarantees consensus when honest users own more than 2/3 of the total user weight. Hence, it is only possible for an adversary to attack the system if they possess more than 1/3 of the total number of coins.

- Using committees: To scale the system, Algorand uses committees to reach consensus in each step of the protocol. In each step, it uses the BA* to randomly select a committee, i.e., a small set of users, based on their weights. Then, committee members must broadcast their messages to other peers in the network. To avoid the possibility of a targeted attack on a committee member, BA* uses cryptographic sortition, which lets users determine their eligibility to join a committee by checking if they obtain a hash value below a certain target.

- Participant replacement: An adversary may choose to attack a committee member after they broadcast a message. To avoid targeted attacks, BA* replaces committee members in every step of the protocol. BA* does not keep any private state, and once a committee member speaks, they immediately become irrelevant to BA*.

Algorand requires a weak synchrony assumption for safety, where the network can be asynchronous for a long but bounded period. To achieve liveness, Algorand requires a strong synchronous assumption and assumes that most of the honest nodes communicate with each other in a bounded time. Under this assumption, Algorand prevents an adversary from creating network partitions or controlling a group of honest users. To recover liveness, Algorand assumes all users have loosely synchronized clocks so that they can kick off the recovery at approximately the same time.

When a user receives some messages, it broadcasts those messages to the surrounding peers via a gossip protocol. To initiate each round, all peers run the cryptographic sortition algorithm to create a committee. Each member of this committee proposes one block. Other users will wait to receive the proposed blocks and only keep the block with the highest priority. The priority of a block is obtained by hashing the output hash h of the VRF function concatenated with the sub-user index. After that, the users who received some blocks will initiate BA*. If an adversary proposes a block with the highest priority in a round, they can propose an empty block and prevent any transactions from being confirmed. However, this happens with a probability of at most $1 - t$, where t is the proportion of honest users. By Algorand's assumption, at least $t > 2/3$ of the weighted users are honest. BA* runs in 2 phases. The first phase is called Reduction, where the committee members reach consensus on either a proposed block or an empty one. The second phase is BinaryBA(), where all users decide on either an empty block or the block

passed from the reduction phase. However, a major drawback of Proof-of-Weight is the lack of incentive and its strong synchrony assumption for liveness. It does not give participants any reward to participate in a committee.

2.2.2 Permissioned Blockchain and BFT Protocols

As shown in the previous section, to preserve the anonymous property of a permissionless blockchain, its consensus protocol has to be resilient against Sybil attacks, which creates some overhead. For example, a Proof-Of-Work scheme requires a large amount of computational power. In June 2018, the yearly energy consumption of Bitcoin was estimated to be between 15 and 50 TWh [38]. Another approach is using a Proof-Of-Stake scheme which is used by Ethereum 2.0 [24] and Algorand [27]. In Proof-Of-Stake, there is a group of validators that mine the blocks. To determine the mining power, a validator "stakes" its coins to validate a block. The more stakes it gives, the higher chance it gets the right to decide the next block to append to the chain. When an attacker tries to validate an invalid block, it loses its stake and is not allowed to validate future blocks. To avoid the richest member having centralized power, there are different variants to the Proof-Of-Stake scheme, such as randomized block selection and coin-age-based selection [33, 35]. A Proof-Of-Stake protocol generally requires less power consumption, provides higher throughput and lower latency than Proof-of-Work. However, it can be challenging at bootstrapping to convince a new validator about the validity of the chain [27].

Unlike permissionless blockchains, in a permissioned blockchain, the set of participants is identified and controlled. A permissioned blockchain can be a suitable fit where participants are willing to share their identities, although they may not need to trust each other. In the next section, I will take a look at RedBelly [17], a promising permissioned blockchain under development, and RCanopus [32], an ordering service for permissioned blockchains. Some notable permissioned blockchains that are not discussed in this thesis include HyperLedger Fabric [3] and Ripple [67].

RedBelly

Red Belly Blockchain [17] is a blockchain system built on top of the consensus protocol DBFT [16]. It allows transactions submitted by public nodes, and a set of private nodes aggregate them into blocks and decide their order. Unlike traditional BFT protocols, it can scale up to 1000 processes across different regions and claim a transaction finality within 3 seconds. With 1000 processes across 14 regions on AWS, it achieves up to 30000

transactions per second with a latency around 3100ms [17]. For future work, Red Belly should be able to identify and punish misbehavior and gives rewards to honest nodes.

The set of n private nodes that participate in validating blocks is recorded in the genesis block, and they reach consensus using the DBFT protocol. The public nodes propose the transactions to the blockchain through the n private nodes. At every round, a subset of n' nodes, named a *configuration*, participate in the consensus protocol. The nodes in this configuration validate transactions using the ECDSA algorithm [31], and they may reach agreement on reconfiguration and renew the memberships of the subset.

The core of Red Belly Blockchain is its leaderless consensus protocol, DBFT [16]. In DBFT, the participating nodes run asynchronously and sequentially, meaning they process requests at their own speed and do not know about the progress of the others, but execute the steps in the same order. When nodes run asynchronously, the system achieves the safety property. To ensure the consensus termination property (i.e., liveness), the system operates under a partial synchrony assumption: After a finite time, there is an upper bound on message transfer and process computation delays. When there are n nodes in the system, it is secure when there are at most $t < \frac{n}{3}$ Byzantine failure nodes. In DBFT, the authors introduced a binary Byzantine consensus $Psync$ where the decision is either True or False. $Psync$ does not use signatures or randomization, terminates in $O(1)$ message delays when all non-failing processes propose the same value, and in $O(t)$ message delays otherwise. DBFT relies on a reduction from multivalue consensus to Psync to build a consensus algorithm for blockchains.

RCanopus

RCanopus [32] is a scalable, leaderless distributed consensus protocol that ensures that live nodes in a system agree on an ordered sequence of operations. This can be used as an ordering service for permissioned blockchains such as HyperLedger [3]. This work is an extended version of Canopus [65] that deals with Byzantine failures. The key idea is to use a virtual tree overlay for organizing participants, thus disseminating messages to limit network traffic across oversubscribed links. This hierarchical structure helps break down the consensus into smaller independent chunks where it is more efficient to reach consensus. To achieve safety and liveness, RCanopus requires synchrony within datacenters and weak synchrony across datacenters.

There are 2 layers in RCanopus: SuperLeafs (SL) and Byzantine Groups (BG). Each SL lets a leader participate in the BFT consensus protocol and replicate the result to the

other nodes within its SL. A BG is a group of several geographically proximate SLs. BGs enable RCanopus consensus to execute in parallel and thus improves throughput.

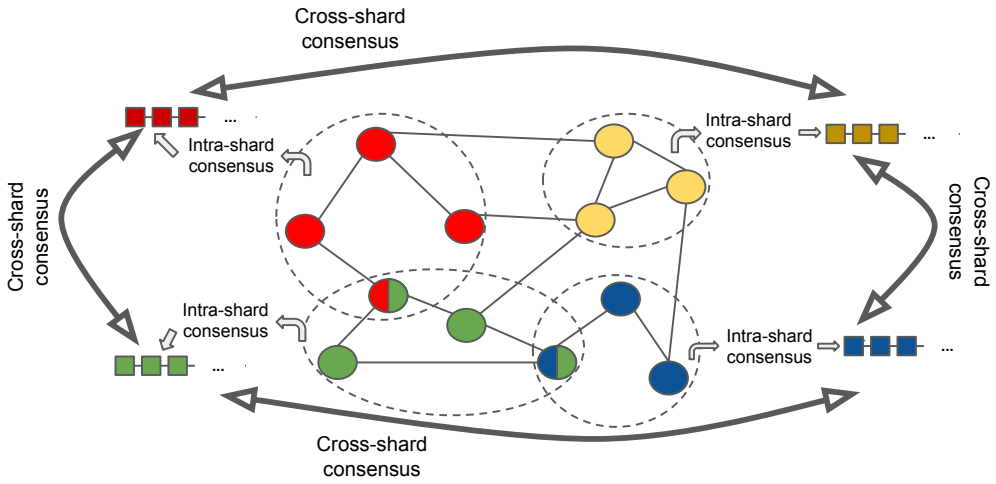
The RCanopus protocol runs in cycles. During a consensus cycle, the protocol determines the order of pending transaction requests and performs the requests in the same order at every node in the group. There are 3 phases in each cycle. In round 1, nodes within each SL reach consensus on a transaction block. Clients send transactions to the servers in SLs, and servers group those transactions into a block. Then, each SL elects a server as a leader to participate in round 2. In round 2, the elected SL leaders commit the transactions within each BG using an existing BFT protocol such as PBFT [10], BFT-SMART [6], or SBFT [28]. In a system with $n = 3f + 1$ processes, these protocols allow f Byzantine failures and thus require at least $2f + 1$ SL leaders to commit a block. Round 3 is the exchange of transactions across remote BGs. The SLs in one BG independently fetch BFT results from BGs in different regions and replicate the results within each SL.

For an RCanopus system to scale well, it is important to design the leaf-only tree carefully. The time it takes to complete the operations within an SL is less than the round-trip time between SLs. In the RCanopus paper [32], the authors discussed several possible faults that RCanopus can tolerate, such as Byzantine fault.

2.3 Sharded Blockchain Protocols

Figure 2.3 describes a typical sharding mechanism where each color represents a shard. In some protocols, one node can participate in multiple shards. The nodes in each shard accept transactions from clients, propose blocks and reach agreement using an intra-shard consensus, generally a Byzantine Fault Tolerant (BFT) [42] protocol. Ideally, the nodes in each shard are geographically proximate to each other but located in different data centers to increase the efficiency and safety of each shard. Then, for transactions that require cross-shard validation (e.g., a transaction whose inputs are from two shards), the protocol requires a cross-shard mechanism to ensure there is no double-spent transaction.

Recent works have been focusing more on sharding in permissionless blockchain systems. An example of a sharded permissioned blockchain is RSCoin [19]. However, it was designed to work with centralized banking systems, relied on a centralized point of authority, and did not consider the Byzantine environment. In general, there is more study for sharding in permissionless blockchains since the network is often geographically distributed and the number of participants is big. In this section, I summarize some of the most well-known sharding techniques used in permissionless blockchains and discuss their scalability, performance, and issues.



The nodes are partitioned into 4 shards represented by 4 colors: red, green, blue and yellow. In some sharding protocols, it is possible for one node to participate in multiple shards. Each shard runs an *intra-shard consensus* to compute a blockchain within each shard, and the protocol requires a mechanism to validate *cross-shard* transactions.

Figure 2.3: A typical sharding mechanism in blockchain.

2.3.1 Elastico

Elastico is a distributed agreement protocol for permissionless blockchains [50]. Elastico aims to solve the scalability problem by merging two paradigms: sharding and Byzantine fault-tolerant transactions. It is the first secure sharding protocol in Byzantine settings. In Elastico, incoming transactions are partitioned into shards. Each shard is verified in parallel by disjoint committees of nodes.

Elastico proceeds in *epochs*, each *epoch* has 5 steps: ① Identity establishment and committee formation; ② Overlay setup for committees; ③ Intra-committee consensus; ④ Final consensus broadcast; ⑤ Epoch randomness generation. Nodes can not be allowed to choose a committee to join since this would allow adversaries to overpower some particular shards. In the first step, to join the network, a node must solve a Proof-Of-Work puzzle, and then their identity is established. Each node must find a nonce, similarly to what the miners do in Bitcoin [61]. The ID is derived from a hash function as follows: $ID = H(EpochRandomness, IP, Public\ key, Nonce) < D$, where D is the difficulty, $EpochRandomness$ is a random number generated every epoch by the consensus committee, the IP and public key are inputs from the node, and the nonce is the puzzle that the

node must solve. Then, Elastico fairly distributes identities to committees using the last k bits of their *ID*. In step 2, to broadcast all identities, Elastico takes a hierarchical approach and uses directory committees. The first C identities become directory servers, where C is the number of committees and hardcoded in a global configuration file. Each committee must have at least C members. The other nodes send their identities to directories and directories broadcast the list of identities. This way reduces the number of messages from $O(N^2)$ to $O(NC)$. In step 3, once the committees are established, an existing BFT protocol such as PBFT is used to reach consensus within each committee (i.e., intra-shard consensus). Each transaction requires signatures from more than half of the committee members and then the value is sent to the final consensus committee. In step 4, a final committee is formed. Committee members in the final committee verify every transaction using a Byzantine consensus protocol and in step 5, the final committee runs a distributed commit-and-xor scheme to generate *EpochRandomness* for the next epoch. The downside of Elastico lies in the unscalability of PBFT as the intra-shard consensus protocol and the weakness of the commit-and-xor scheme to generate *EpochRandomness* [78], which are improved in OmniLedger and RapidChain.

2.3.2 OmniLedger

OmniLedger aims to be a secure permissionless distributed ledger that provides scalability and performance on par with centralized payment systems such as Visa [37]. OmniLedger consists of multiple shards, combining different techniques such as RandHound [70], cryptographic sortition [27], ByzCoinX (OmniLedger's enhanced version of ByzCoin [36]), and Atomix (OmniLedger's novel two-step atomic commit protocol) to ensure security and correctness within and across shards while achieving high performance and scalability.

Validators are nodes that verify there is no double-spent transaction in blocks. To securely shard validators, OmniLedger uses a global identity blockchain and a distributed randomness generation protocol. Similarly to what happens in step 1 of an Elastico epoch (see Section 2.3.1), validators who wish to join OmniLedger have to first register to the identity blockchain. Their identities and respective proofs are broadcast on the gossip network. Similarly to Elastico, validators themselves can not be allowed to choose a shard to join. When validators are assigned randomly, with high probability malicious nodes are distributed evenly in all shards. In OmniLedger, the source of randomness comes from RandHound [70] and cryptographic sortition. At the beginning of each epoch, OmniLedger uses RandHound to assign new validators and reassign existing validators to new shards and groups within shards. The number of groups in each shard is specified in a shard policy file. RandHound is a large-scale distributed protocol that "provides publicly-verifiable,

unpredictable, and unbiasable randomness against Byzantine adversaries." RandHound requires a leader, so OmniLedger utilizes cryptographic sortition, which is based on verifiable random functions VRFs [55], to randomly select a leader at the beginning of each epoch. The leader requests a collective signature and appends the block to the identity blockchain if the block is valid (i.e., endorsed by at least 2/3 of the validators).

Each shard runs ByzCoinX internally to reach consensus. ByzCoinX is a protocol based on ByzCoin [36] with enhanced performance and robustness. ByzCoin uses a tree communication pattern to distribute blocks and provides a slow-path for fault-tolerance. OmniLedger introduces a new communication pattern to increase robustness but trades off some scalability. OmniLedger modifies the message propagation mechanism and chooses a leader in each group. Each group leader is responsible for managing communication between the protocol leader and the respective group. During the setup described above, validators are evenly assigned to not only every shard, but also each group within the shards. Transactions that do not conflict and have no UTXO dependencies can be processed in parallel. To process transactions across shards atomically, OmniLedger uses a Byzantine Shard Atomic Commit protocol named Atomix. In the first phase (Initialize), a cross-shard transaction is created and gossiped on the network, and eventually reaches all input shards. In the second phase (Lock), each input shard validates the transaction to ensure the input has not been spent. In the third phase (Unlock), the client unlocks-to-commit if all shards accept the transaction and unlocks-to-abort otherwise.

Compared to Elastico, OmniLedger provides a more secure mechanism to assign nodes to shards, proposes an atomic protocol for inter-shard communication, and further reduces communication overheads. However, there are several problems that OmniLedger has yet to solve. OmniLedger requires a trusted setup to generate an initial configuration to seed VRF. Clients must actively participate in cross-shard transactions, and malicious clients can cause infinite blocking in the lock and unlock phases.

2.3.3 Rapid Chain

RapidChain is the first full-sharding-based permissionless blockchain protocol that is Byzantine fault-tolerant and requires no trusted setup [79]. It achieves high throughput and good scalability by using a novel intra-committee consensus algorithm. RapidChain has 3 main phases: Bootstrap, Consensus, and Reconfiguration. The bootstrap phase is executed only once at the beginning. Then, RapidChain proceeds in epochs, each epoch consists of one Consensus phase and one Reconfiguration phase. RapidChain bootstraps by choosing a root group that consists of $O(\sqrt{n})$ nodes, where n is the total number of nodes. This group

creates and distributes a sequence of random bits used to establish a reference committee of size $O(\log n)$. Then, the reference committee creates k random committees (shards), each committee has $m = c \log n$ members where n is the number of nodes, and c is a security parameter typically set to 20. In every epoch, nodes that want to join or stay must solve a Proof-Of-Work puzzle, which is randomly generated every epoch. The referenced committee verifies their solutions, produces a reference block with the list of all active nodes and their committees, and sends the block to all other committees. To make reconfiguration secure and protected against a slowly-adaptive Byzantine adversary, RapidChain builds on the Cuckoo rule [68], which is a selective random shuffling mechanism. This makes it hard for adversaries to target a specific committee. Once committees are formed (or reformed), within each committee, an intra-committee consensus protocol is used to reach consensus. The protocol is built on a gossiping protocol to propagate the messages (e.g., transactions and blocks) among committee members, and a synchronous consensus protocol to agree on the header and hash of the block. The gossiping protocol is derived from the Information Dispersal Algorithm (IDA) and the erasure coding mechanism. A large message is divided into chunks, including one parity chunk. A Merkle-tree is built and uses the chunks as leaves. Each neighbor receives a unique subset of chunks and their Merkle-proofs. The IDA-Gossip protocol reduces communication overhead and is faster than reliable broadcast protocols. However, it is not reliable and requires a consensus protocol to run on the root to achieve consistency. The consensus protocol used in RapidChain is a variant of Abraham et al.'s Efficient Synchronous Byzantine Consensus [1].

Transactions whose inputs and outputs are from different committees require cross-shard verification. The input committees of a transaction store the inputs of the transaction and the output committees are chosen based on the hash of the transaction id. Cross-shard communication in RapidChain uses the Kademlia routing algorithm. Each node stores information about their committee members and $\log \log n$ nodes in each of their $\log n$ closest committees. When a message arrives, all nodes in the sender committee send it to all nodes they know. Each receiver invokes the IDA-gossip protocol to send the message to their committee members. Compared to Elastico and OmniLedger, RapidChain does not require a trusted setup and further improves the performance. However, it still relies on synchrony for liveness and there is no incentive mechanism for active nodes.

2.3.4 Learnings

As we observe from the sharding protocols in Section 2.3, BFT protocols for permissionless blockchains such as Proof-Of-Work and Proof-Of-Stake are widely used to establish identities and allow nodes to join/rejoin the network. In order to maintain the security

of a sharded blockchain system, it is essential to prevent shard takeovers. A node cannot be allowed to choose a particular shard to join, otherwise, malicious nodes can take over a shard and corrupt the corresponding portion of data. Therefore, randomness generation protocols such as RandHound are used to distribute nodes randomly. As a result, the percentage of malicious nodes in each shard is roughly equal. Once shards are established, BFT algorithms are used to reach consensus and validate transactions within and across all shards.

Cross-shard transactions are a major factor that creates the overhead of data transmission among shards, thus they degrade the system throughput and increase confirmation latency. Sharded blockchains have two major approaches to handle cross-shard transactions. The first approach is to build a full-mesh connection among nodes (e.g., OmniLedger [37] and RapidChain [79]). Verifying a cross-shard transaction requires communication between the validators in different shards. OmniLedger proposes the *Atomix* protocol, where clients are in charge of exchanging cross-shard transactions. Clients must get *proof-of-acceptance* from all of the input shards to lock the input transactions and send the transactions and their proofs to the output shards. RapidChain optimizes the *Atomix* protocol by proposing a three-way confirmation mechanism. Each committee in each shard maintains a routing table of $\log_2 n$ committees to improve the communication among shards. This approach, although it spreads out the storage and avoids a computation bottleneck in the main chain, can bring communication overhead and security challenges. It generally requires nodes to periodically reshuffle into new shards and download the ledger of the new shard they are being reshuffled to. Since the transactions are separated into different chains, if a shard is controlled by an adversary, other shards can no longer validate transactions that have dependencies on the attacked shard.

Another approach is to store a global root chain, such as in Elastico [50] and the ongoing Ethereum 2.0 [25]. After the nodes in each shard finalize and reach consensus on local transactions, some of the nodes will send them to the final committee, and the final block is stored in a global ledger. In Ethereum 2.0, the global chain - the beacon chain - ensures that the transactions in all shards are in sync and discards any double-spent transaction. The shards increase parallelism, while the root chain handles cross-shard transactions. Comparing to the first approach, this approach reduces the overhead of data migration when handling cross-shard transactions, and the system maintains correctness even when one shard is controlled by an adversary. Besides, nodes can join shards without the need to reshuffle.

The second approach brings a question of how to combine the blocks in different shards to obtain the main chain. In Chapter 3, I will examine two approaches to interleave blocks, referred to as *strong temporal coupling* and *weak temporal coupling*.

Chapter 3

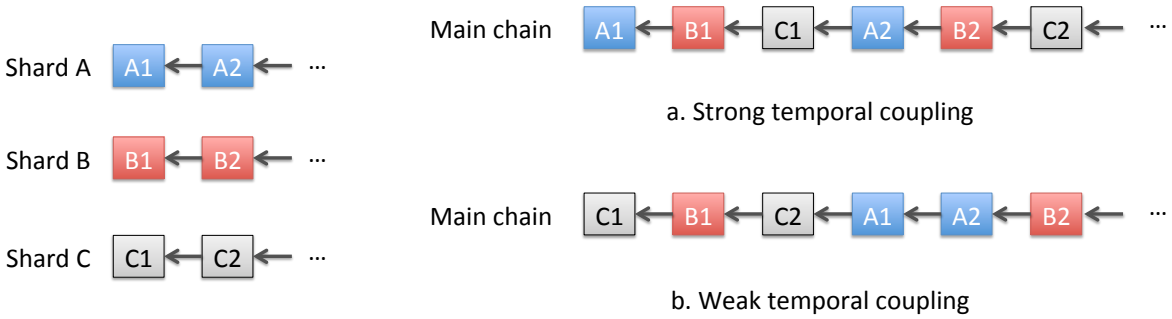
Interleaving Blocks in a Sharded Permissioned Blockchain[1]

As discussed in Section 2.3.4, it is important that a sharded blockchain can efficiently support cross-shard transactions. A notable approach is to build a global main chain replicated among validators (e.g., Ethereum 2.0 [25], Elastico [50], and RCanopus [32]). Each shard independently maintains and processes disjoint subsets of transactions, while the main chain maintains the total order of transactions.

In this chapter, I discuss two different methods to interleave blocks from different shards to build a global main chain, referring to them as *strong temporal coupling* and *weak temporal coupling*. This chapter is organized as follows. Section 3.1 explains the difference and trade-offs between the two methods of interleaving. Section 3.2 discusses the design of our shared permissioned blockchain[2] where we examine the performance of interleaving blocks using *strong* and *weak temporal coupling* methods. We use EPaxos [58] as the replicated ordering service for the blockchain. This blockchain system follows the UTXO format, which is similar to what Elastico, OmniLedger, and RapidChain uses and different from the account balance model in Ethereum 2.0. Section 3.3 discusses in detail the implementation decisions and the changes that I have made compared to the key-value store application in EPaxos [58]. Finally, in Section 3.4, I discuss the set-up of the experiments to compare the performance of the two interleaving methods.

[1] Some parts of this chapter were developed jointly by me, Mr. Mao and Dr. Golab for a poster paper at ICBC 2020 [51]

[2] The main design ideas were influenced by RCanopus [32] and BoscoChain [72]



In strong temporal coupling, nodes predetermine the order of blocks. In weak temporal coupling, nodes interleave blocks dynamically.

Figure 3.1: Methods of interleaving

3.1 Methods of Interleaving

Strong temporal coupling refers to the scenario where all the nodes interleave blocks in a fixed, round-robin order. An example of strong temporal coupling is RCanopus, where the nodes decide the global order of transaction blocks using a cycle-based consensus [32]. As described in Figure 3.1, suppose we have three shards A, B, C, and each has their chunk of blocks. In strong temporal coupling, every node agrees on the order of the shards. For example, in Figure 3.1, the order is A, B, C. Each node pulls blocks in cycles, and in cycle i, each node pulls the i-th blocks from all the shards in the predetermined order. On the other hand, in the weak temporal coupling, the blocks are interleaved dynamically. All the nodes must have an additional layer of communication to reach consensus on the order of blocks.

The strong temporal coupling approach can guarantee consistency if there is no Byzantine failure. All the nodes can interleave independently without communicating with other peers. However, the system stops making progress if a shard becomes unavailable or stops growing. On the contrary, the weak temporal coupling method can guarantee consistency even if some shards are unavailable. However, it requires an additional layer of consensus.

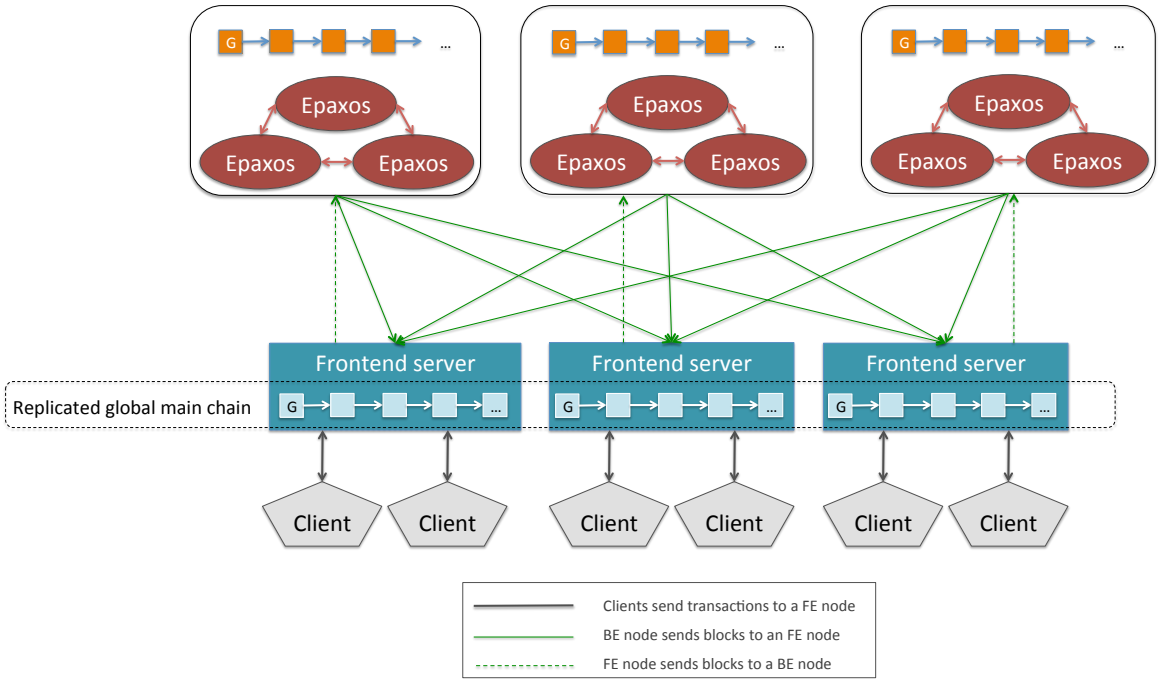
Using the strong temporal coupling method requires the system to keep track of the growing rates of the shards or another mechanism to prevent the system from stalling. Ethereum 2.0 [24] and Elastico [50] use a weak temporal coupling mechanism to order blocks from different shards.

3.2 System Architecture

We design a prototype in Golang to compare the performance between strong and weak temporal coupling [51]. There are three layers of components in our system architecture: Clients, Front-end (FE) servers, and Back-end (BE) servers, as shown in Figure 3.2. Clients send transactions to front-end servers. Front-end servers verify and aggregate transactions, then they submit blocks of transactions to back-end servers. We have multiple groups of back-end servers, each group acts as a shard, and each shard reaches consensus using EPaxos [58, 59]. Once each shard receives blocks from front-end servers, it fills the timestamp field of the block header, then EPaxos orders and replicates blocks. Then the blocks are sent to and interleaved by the front-end servers. In strong temporal coupling, the order of shards is predetermined and all front-end servers interleave the blocks in that order. In weak temporal coupling, the front-end servers reach consensus on the order of blocks using EPaxos (Figure 3.2).

Current blockchains depend on expensive consensus mechanisms to achieve consistent replication. These consensus mechanisms pose performance bottlenecks due to sacrificing performance for Byzantine fault tolerance [14]. As analyzed in [29], turning off PBFT in Hyperledger Fabric and Proof-of-Work in private Ethereum, the throughputs are almost doubled, and the latency is reduced by half. To show the difference between the two methods of interleaving, we pick EPaxos as the ordering service. EPaxos is an open-source, leader-less Paxos-based algorithm that achieves crash-fault tolerance. An EPaxos shard with three replicas can process up to 25000 1KB-long requests per second when state machine replicas and clients use two 64-bit virtual cores with 2 EC2 Compute Units each and 7.5 GB of memory [58, 59]. Although EPaxos is not Byzantine fault-tolerant, by reducing the bottleneck of the consensus protocol, we expect to observe a larger gap in performance between the two methods of interleaving. We also encapsulate the services provided by the back-end and front-end nodes so that in the future works, we can feasibly switch to a different ordering service.

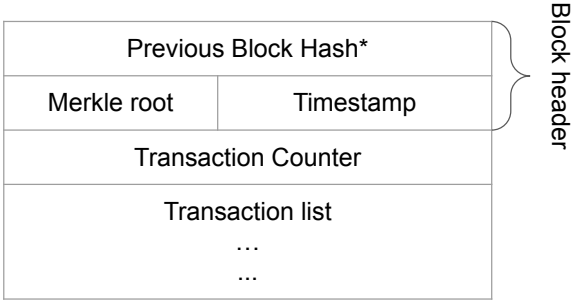
We follow the block and transaction format of Bitcoin with some simplifications, as described in Figures 3.3 and 3.4. There is no chain code or personalized script, and each transaction is a transfer of coins from one user to another. In Bitcoin, a sender can send a portion of a coin to a receiver and the remainder goes to themselves. The sender may have to pay some transaction fee to the miner. In our system, there is no transaction fee, each transaction has one input and one output, and each transaction represents the transfer of one coin from the input address to the output address. This is a simplified transaction model but sufficient for comparing strong and weak temporal coupling methods. Similar to Bitcoin, the block id is included in coinbase transactions. Since all the fields of coinbase



Each back-end node has a chain containing a subset of transactions; the first block of each chain is denoted as G. Three back-end nodes form a shard that runs EPaxos to replicate the chain. After front-end servers receive blocks from shards, they validate the transactions. Front-end servers interleave blocks using either *strong temporal coupling* or *weak temporal coupling* methods to obtain the global main chain.

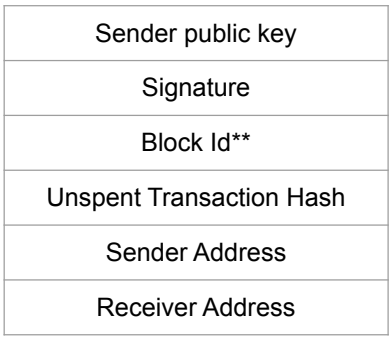
Figure 3.2: System design for comparing methods of interleaving.

transactions sent to a receiver are the same, the block ids are necessary to differentiate the hashes of coinbase transactions. The coin is an unspent transaction, and only the owner can spend it. Each user has an ECDSA public/private key pair and is identified by a unique address, which is the SHA-256 hash of the user's public key. Each block consists of a timestamp, a previous block hash, and a list of at least 1 transaction and up to 1000 transactions. The timestamp is assigned by the shard when it receives the block, and the previous block hash is the hash of the most recent block in the global root chain. Each transaction contains the information sent by the sender S, including their (a) public key, (b) digital signature, (c) address, (d) the input unspent transaction hash, and (e) the receiver R's address. We use the ECDSA algorithm and S's private key to sign (d), generating (b) for validation. We keep a hash map of all the unspent transactions, and each



*The hash of the most recently added block (i.e., the tail block) of the global root chain

Figure 3.3: Block structure



** Block id is included in coinbase transactions

Figure 3.4: Transaction structure

transaction is validated before it is appended to the main chain of blocks. We use function `validateInputs()` (see Figure 3.5) to make sure that the inputs of the transactions are not yet spent and there is no double-spent transaction within each block. Using (a), (b) and the unspent transaction map, we verify that S is the receiver of (d), the transaction is unspent, and (b) is signed by S using ECDSA. If all of these conditions are satisfied, we append the transaction to the main chain of blocks. Otherwise, we discard the transaction. This transaction model is termed as an unspent transaction output model, short for UTXO, which is commonly used in blockchain systems [3, 26, 61].

Each client, front-end, and back-end node has a host and a port number, and nodes communicate with each other using `gRPC`. A client node is designed to mimic multiple users. Each client has multiple `goroutines` (lightweight threads in Go), where each `goroutine` represents a user trying to send a coin to another user in the same client. Each `goroutine` stores a pair of private and public keys, and a pool of UTXO hashes. As in Bitcoin, the address of a user is defined as the hash of their public key, alongside a version and a prefix. Currently, in Bitcoin, this prefix is either 1 or 3, which respectively represent Pay-To-Public-Hash and Pay-To-Script-Hash types of transactions [60]. Each `goroutine` acts as a sender, pulls an unspent transaction hash from its pool of UTXO hashes to generate a transaction, sends it to an FE server, and waits for a response. After a transaction is validated, the FE server reports whether the transaction is appended successfully to the main chain, or discarded. If the transaction succeeds, the FE server sends the transaction hash to the sender, and the transaction hash is pushed to the pool of UTXO hashes of the receiving user. After processing the response, the sender continues spending more coins.

In the prototype, the sender and the receiver are always in the same client process to minimize the communication overhead among different clients.

Each FE server acts as an intermediary between clients and BE servers. Each FE server concurrently receives and saves the input transactions from multiple clients. It batches and inserts the transactions into a block, and then sends the block to the closest shard. Each shard is implemented using an EPaxos state machine with three replicas, where each replica is a BE server. Once a quorum of BE servers in a shard reach consensus, modify the block header and store the block, one of them sends the response to the FE server. Meanwhile, some concurrent `goroutines` in the FE server continue pulling the blocks stored in shards. For the strong coupling method, another `goroutine` in the FE server interleaves the pulled blocks in a fixed order. For the weak coupling method, the FE servers run another EPaxos state machine for interleaving blocks. Each FE server proposes the pulled blocks and waits for the other FE servers to agree on its proposal. Once the FE servers reach consensus, the block is confirmed and appended to the main chain. In both cases, all FE servers compute the same main chain of blocks.

3.3 Implementation

In EPaxos [58] and their open-source code base [59], the authors evaluate the protocol in the context of a key-value store. Clients send updates (read or write) requests to the servers, which replicate the key-value store using EPaxos. In this section, I discuss the changes I have made to EPaxos to make it feasible to replicate blocks and transactions, as well as provide the pseudo-code of the system.

3.3.1 Protocol Buffers and gRPC

EPaxos uses a custom message encoding with `net/rpc`, the Go standard library, for the communication among nodes within a shard and the client-server communication [59]. In `net/rpc` library, each request is independent, where a client writes one request to the server and receives one response. Even though `net/rpc` is easy to set up and allows customized message encoding, I observe that this is not a good fit for our blockchain system and creates performance bottlenecks. Unlike the key-value store example in EPaxos, each client in our blockchain protocol continuously sends transactions to a front-end server. Front-end servers and back-end servers frequently exchange messages: Front-end servers periodically batch transactions into blocks and send them to back-end servers, while back-end servers

distribute blocks to front-end servers. We need an RPC protocol that allows data streaming to improve the utilization of connections. `net/rpc` does not support streaming, so I change the protocol to `gRPC` (general Remote Procedure Call), a high-performance RPC framework developed at Google.

`gRPC` is a remote procedure call running over HTTP/2.0 [47]. It allows services to communicate and run functions on different machines efficiently. When a client wants to call a method in a server, it creates a local object that stubs the implementation of the methods of the corresponding service in the server. The client sends a `gRPC` request with the method name and parameters, wrapped inside a protocol buffer message, to the server. After decoding the client request, the server processes the client call, executes the service method, and possibly sends the result back to the client. Compared to `net/rpc`, `gRPC` supports `net/context` and `net/trace`, which in turn allow clients to cancel pending requests, set timeout and deadline, and trace RPC requests and long-lived objects [46]. Besides, we can easily change the fields of the objects without the need to implement customized message encodings.

`gRPC` is compatible with various serialization formats, including Thrift, JSON, and Flatbuffers [47]. However, in this design, I choose to use Protocol Buffer [48], a serialization mechanism developed by Google and the default Interface Definition Language (IDL) of `gRPC`. Compared to other formats such as JSON, protocol buffer provides a lightweight message in binary format. Protocol buffer automatically generates client and server-side stubs after we define services in a `.proto` file. I implement all client requests and server responses as protocol buffer objects.

`gRPC` can provide efficient processing service through multiplexed streams, where multiple requests can be processed with a single connection. `gRPC` supports four types of service methods: ① Unary RPCs: Similar to a typical function call, clients send a single request and receive a single response. ② Server streaming RPCs: Clients send a single request to the server. `gRPC` establishes a one-directional stream from the server to the client where the server keeps sending messages to the client. ③ Client streaming RPCs: `gRPC` establishes a one-directional stream from the client to the server where the client continues sending messages to the server. The server reads incoming requests and returns a response when there are no more messages (e.g., after receiving an `EOF` message). And ④ bidirectional streaming RPCs: A bidirectional stream is established between a client and a server, which allows both sides to read and write messages independently. In our blockchain protocol, I set up a bidirectional stream between each user and front-end server, and let back-end servers send blocks to front-end servers via a server streaming RPC. In Chapter 4, I use a unary RPC to establish connections between full and lightweight front-end servers, a server streaming RPC to distribute blocks metadata and Bloom filters from full front-end

servers to lightweight front-end servers, and a bidirectional stream for lightweight front-end servers to request blocks from full front-end servers.

3.3.2 Concurrency in Go

Go is a programming language developed at Google in 2009 that aims to solve problems that software engineers at Google encounter in their daily works [45]. Go reduces code build time, provides easy-to-understand code, and famously supports concurrency via a built-in mechanism named `goroutine` and a built-in type `channel`. `goroutine` are similar to threads, but more lightweight and easy to create as we can execute thousands of `goroutine` on a single core. Go `channel` works as a synchronized FIFO queue, which multiple goroutines can push to and pull from concurrently. Go `channel` works well as a synchronization primitive for intra-process communication among Go `goroutines`.

As discussed in Section 3.3.1, we mimic the behaviors of multiple users by providing each user a `goroutine`. We control the number of users and the throughput of the system indirectly by adjusting the number of `goroutines` in each client program. For each user, I use a Go `channel` to store their unspent transaction hashes (UTXHash). When a user S sends a transaction tx to user R, S proposes tx to the front-end server. After it is appended to the main chain, S pushes the hash of tx to the corresponding UTXHash Go `channel` of the recipient R. The efficiency of using Go `channel` reduces the overhead of exchanging UTXHashes between users on the client-side.

3.3.3 Functionalities in Front-End Servers

The ease of concurrency control in Golang brings intuition to separate the functionalities of a system into multiple goroutines. Each front-end server has four tasks that can run in parallel, and they can communicate with each other using Go `channel` objects. The four main tasks are:

1. Receive and verify signatures of proposed transactions from multiple clients. Once the transaction is appended to the main chain, send a response with the transaction hash to the client.
2. Aggregate transactions from multiple users to form blocks and send the blocks to a back-end server.
3. Download blocks from one or more shards.

(4a) Interleave downloaded blocks in a pre-determined order to form the main chain (i.e., strong temporal coupling), or

(4b) Propose downloaded blocks to other front-end servers to form the main chain (i.e., weak temporal coupling).

Task (1) is an RPC method which is defined in a `.proto` file. When a client calls this method via RPC, it kicks off a `goroutine` to handle the client connection. If transactions have valid signatures, they are pushed to a `channel` c_0. Periodically, task (2) pulls transactions from `channel` c_0 to form and propose a block to a front-end server. Meanwhile, task (3) runs in k `goroutines` to download blocks from shards and push them to `channels` $c_1, ..., c_k$, where k is equal to the number of shards from which it downloads. Either task (4a) or task (4b) runs in a concurrent `goroutine`, pulling transactions from the downloaded `channels` $c_1, ..., c_k$ to form the main chain. Task (1) and task (2) are the producer and subscriber of the `channel` c_0 of proposed transactions, while task (3) and task (4) are the producer and subscriber of `channels` $c_1, ..., c_k$ of downloaded blocks.

3.3.4 Pseudo-Code Description

Figure 3.5 describes the pseudo-code of the common functions that both strong and weak coupling FE servers use for functionalities (1) - (3) (see Section 3.3.3). Each FE server has the input of the host and port numbers of BE servers, where the i^{th} back-end server comes from the i^{th} shard. In the `init()` function, the FE server creates connections to BE servers using the given host and port numbers. It generates a new blockchain β that contains a genesis block, a new hashmap H that contains unspent transaction hashes, a channel `q` of transactions pending to the proposed, and starts a server `s` that accepts transactions from clients. As a client sends a `gRPC` request to `s`, they kick off a `receive_tx() goroutine` (Task (1)). Over a single connection, the client can submit multiple transactions through a stream of transactions, and the server replies through of stream of transaction hashes. The client can batch transactions and send them over the stream, and will receive the responses of the transactions at the same time. In the experiments described in Section 3.4, there is no batching in this step and each client only sends one transaction to the stream. The function waits until the transaction hashes appear in H and then informs the sender about the transactions. The sender waits for the response from the function before letting the receiver know the transaction hash. In function `receive_tx()`, the channel `q` works as a synchronized queue to which proposed transactions are pushed. Function `append()` (Task (2)) periodically pulls transactions from `q` to form blocks and sends them to back-end

servers. In `receive_tx()`, for each transaction in stream, we check if the input transaction hash exists in the unspent transaction pool H, and if the receiver address of the unspent transaction matches the given address and public key. Then, we use the ECDSA algorithm to verify if the message, which is the signed utxHash, is signed by the owner of the given public key. ECDSA signature verification creates a performance bottleneck in the system, so the validation step is executed in a `goroutine`, denoted as **go** in the pseudo-code. This helps the `receive_tx()` function continuously receive transactions and allows transactions to be verified in parallel. After initializing Π, β, H, s, and q, the `init()` function kicks off the `append()` function in a `goroutine`. In `append()`, the FE servers can pick the destination BE server (e.g., choose the nearest BE server) or send blocks to different BE servers in a round-robin order. In the experiments described in Section 3.4, the front-end servers send blocks to the nearest back-end server.

The blockchain β is replicated among all FE servers, so all the β in all FE servers have the same genesis block. H stores the hashes of unspent transactions, which is used in Tasks ① and ④ (i.e., functions `receive_tx()`, and either `interleave()` or `execute_proposal()`) to check if a transaction is not yet spent. In Task ④, after a block b is validated to have no double-spent transaction, the input transaction hashes of transactions in b are deleted from H. The FE servers compute the previous transaction hash of block b and appends block b to the main chain. Then they can safely process the next block. In a separated `goroutine`, the hashes of the transactions in b are appended to H.

Given the back-end shard id i, the `get_blocks()` function creates a server-side stream that lets a back-end server in shard i continuously distributes blocks to the FE server, which in turn saves the block to a channel q_i. As described in Figure 3.6, a strong-coupling FE server downloads blocks from all the shards by calling multiple `get_blocks()` `goroutines`. As the `get_blocks()` functions push blocks to channels q_1, q_2, ..., q_n, the `interleave()` function reads blocks from the channels in an order pre-determined by all FE servers. If a block b has no double-spent transaction, it is appended to the main chain β and their transaction hashes are added to H. Meanwhile, as described in Figure 3.7, a weak-coupling FE server downloads blocks from one shard given an id by calling a single `goroutine get_blocks()`. It uses EPaxos to connect with other FE servers and proposes its downloaded blocks. As a weak-coupling FE server receives a block proposal b, it starts the `execute_proposal()` function that validates the block, adds b to β, and adds their transaction hashes to H.

```
1  Function init(L)
       Input : A list of hosts and port numbers L of backend servers
2      Π ← ∅   a global FIFO channel of connections to backend servers
3      foreach host, port ∈ L do
4          con ← TCP connection to backend server at host:port
5          add con to Π
6      end
7      β ← a new global main chain
8      H ← {}   a new global unspent transaction hashmap
9      q ← ∅    a new global transaction channel
10     s ← a new server accepting transactions from clients
11     go append(q, Π)
12 end
13 Function receive_tx(stream, client, q, H)                    // Task (1)
       Input : A stream of transactions from client
               The global transaction channel q
               The global unspent transaction hashmap H
14     local-q ← ∅ a new local transaction channel
15     while (tx ← a transaction from stream) ≠ EOF do
16         utxHash, pubKey, sig, msg ← UTX hash, public key, signature, and
17                                     signed message in tx
18         go:
19             valid ← utxHash ∈ H && utxHash.receiver owns pubKey
20                     && ecdsa.verify(pubKey, sig, msg)
21             if valid = TRUE then
22                 add tx to local-q
23                 add tx to q
24             end
25         end
26     end
27     while local-q not empty do
28         tx ← a transaction in local-q
29         if tx ∈ H then
30             send tx.hash to client
31             remove tx from local-q
32         end
33     end
34 end
35 Function append(q, Π)                                        // Task (2)
       Input : The global transaction channel q
               The global channel Π of connections to backend servers
36     prevTime ← currentTime()
37     while q.size() > 100 or (currentTime() − prevTime) > 100ms do
38         b ← a block created from transactions in q
39         con ← the first connection in Π
40         send b to a backend server using con
41         append con to the end of Π
42         prevTime ← currentTime()
43     end
44 end
```

continued on next page →

```
43 Function get_blocks(i, L, q_i)                                    // Task (3)
       Input  : The backend server id i
                A list of hosts and port numbers L of backend servers
                The global channel q_i containing blocks downloaded from shard i
44     stream ← a server-side gRPC stream using the host and port numbers in L[i]
45     while TRUE do
46         add (b ← a block from stream) to q_i
47     end
48 end
49 Function validateInputs(H, b)
       Input  : The global unspent transaction hashmap H
                A block b
       Output: If block b is valid and has no fake/double-spent transaction
50     m ← {}    a set of input UTX hashes
51     foreach tx ∈ b do
52         in ← input UTX hash of tx
53         if in ∉ H or in ∈ m then return FALSE        // tx is fake/double-spent
54         add in to m
55     end
56     return TRUE
57 end
```

Figure 3.5: Common functions of strong and weak coupling FE servers

3.4 Evaluation

This section presents the evaluation of the blockchain system using strong and weak coupling methods to interleave blocks. I analyze the peak throughput and latency of the system in a simulated environment and on Amazon Elastic Compute Cloud (EC2).

3.4.1 Simulated Environment

I first experiment in a simulated environment using a 20-core Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz. The operating system is Ubuntu 20.04.1 LTS and the Golang version is go1.13.8 Linux/amd64. The back-end, front-end, and client processes run in separated cores, and the network latency is simulated using the `tc` command. There are two settings

Algorithm 1: Pseudo-code of strong-coupling FE servers

```
1  Function main(L, #shards)
       Input  : A list of hosts and port numbers L of backend servers
                Number of BE shards #shards
2      Π, β, H, s, q ← init(L)
3      for i ← 1 to #shards do
4          q_i ← ∅    a channel containing blocks downloaded from shard i
5          go get_blocks(i, L, q_i)
6      end
7      go interleave(β, H, q_1, q_2, ..., q_n, #shards)
8  end
9  Function interleave(β, H, q_1, q_2, ..., q_n, #shards)          // Task 4a
       Input  : The global main chain β
                The global unspent transaction hashmap H
                A list of channels of downloaded blocks q_1, q_2, ..., q_n
                Number of BE shards #shards
10     while TRUE do
11         for i ← 1 to #shards do
12             b ← the next block from q_i
               // If b has a fake/double-spent transaction, discard b
13             if validateInputs(H, b) = FALSE then Go to line 11
               // Delete spent transactions
14             foreach tx ∈ b do
15                 delete (in ← input UTX hash of tx) from H
16             end
               // Append b
17             b.prevTxHash ← the hash of the last block in β
18             append b to the main chain β
               // Add new transactions to H in parallel
19             go:
20                 foreach tx ∈ b do add tx.hash to H
21             end
22         end
23     end
24 end
```

Figure 3.6: Strong-coupling front-end servers pseudo-code

shown in Figure 3.8, one using 3 BE shards and the other using 5. In each setting, the number of front-end servers is equal to the number of shards, and each front-end server has a dedicated client program that continuously proposes transactions. I use `taskset` to assign the number of cores for each component. Each BE server runs in 1 core, each FE server runs in 4 cores, and each client program runs in 2 cores. The number of cores for each

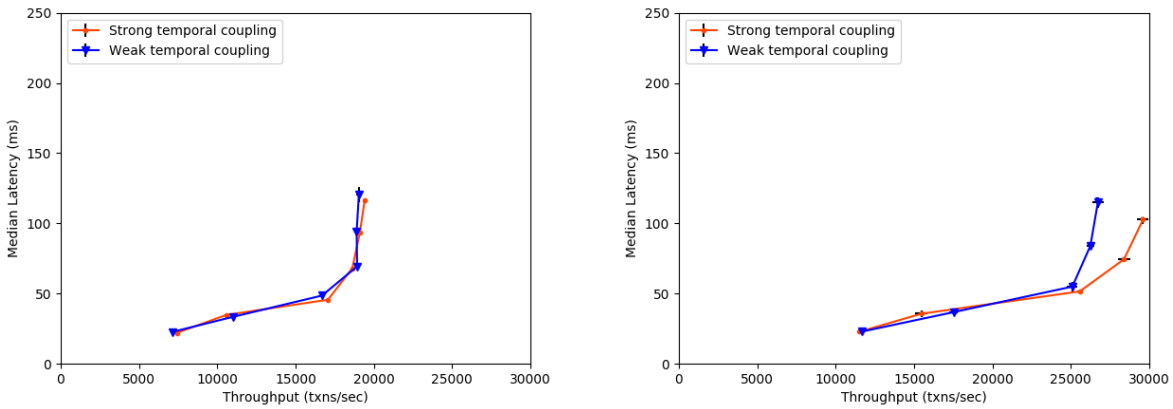
Algorithm 2: Pseudo-code of weak-coupling FE servers

```
   // Each weak-coupling FE server has an id id
 1 Function main(L, id)
       Input  : A list of hosts and port numbers L of backend servers
                The id of the server
 2     Π, β, H, s, q ← init(L)
 3     q_id ← ∅    a channel containing blocks downloaded from shard id
 4     go get_blocks(id, L, q_id)
 5     Γ ← the list of other FE servers connected using EPaxos
 6     while TRUE do
 7         go propose the next block in q_id to Γ          // Task (4b): propose a block
 8     end
 9 end
   // Each FE server in Γ executes this function after receiving b
10 Function execute_proposal(b, β, H)  // Task (4b): process a proposed block
       Input  : A proposed block b
                The global main chain β
                The global unspent transaction hashmap H
11     if validateInputs(H, b) = FALSE then return
12     foreach tx ∈ b do
13         delete (in ← input UTX hash of tx) from H
14     end
15     b.prevTxHash ← the hash of the last block in β
16     append b to the main chain β
17     go:
18         foreach tx ∈ b do add tx.hash to H
19     end
20 end
```

Figure 3.7: Weak-coupling front-end servers pseudo-code

process is chosen by observing the CPU usage of each process using the `htop` command. Each client program has multiple `goroutines`, each `goroutine` mimics the behavior of one user. Each user `goroutine` in the client program has a pair of `ecdsa` private and public keys, which are used to sign and verify transactions. Each `goroutine` has a pool of unspent transactions, from which it pulls and proposes the transactions one by one to the front-end server. At initialization, each user `goroutine` sends coinbase transactions to their pool. After initialization, I record the total number of processed transactions and the end-to-end latency in the client program, and calculate the median latency, as shown in Figure 3.8.

I indirectly control the throughput by changing the number of `goroutines` per client, from 100, 200, 400, 600, 800 to 1000. Each of these numbers is corresponding to a dot in



(a) 3 shards

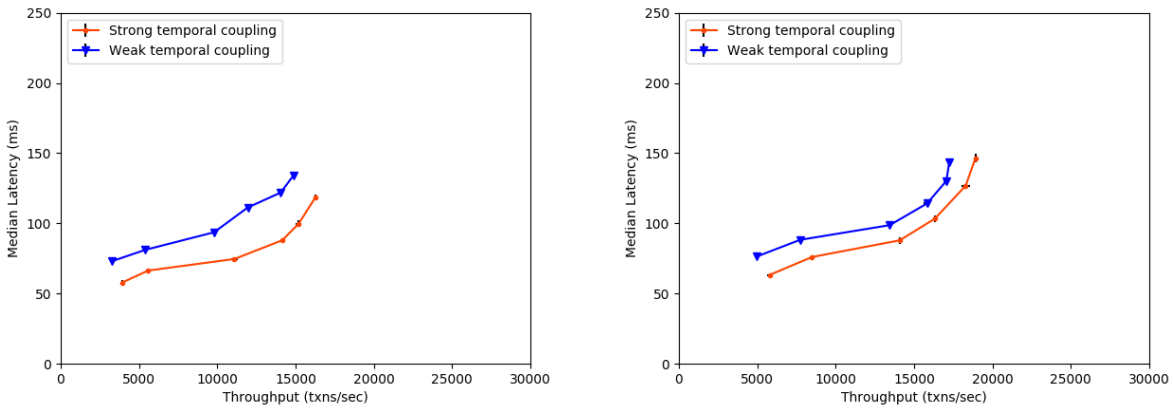
(b) 5 shards

Each dot, respectively from left to right, represents the latency and throughput when each client has 100, 200, 400, 600, 800, and 1000 goroutines.

Figure 3.8: Performance of strong and weak temporal coupling methods in simulated environment with no simulated network latency

Figure 3.8, which shows the trade-off between throughput and latency. For each setting, I conduct the experiment three times, and each run lasts for 20 seconds. The experiment that is shown in Figure 3.8 has no simulated network latency added. The error bar represents the standard deviation of the latency and throughput.

Having three shards (Figure 3.8a), both strong and weak temporal coupling methods reach the same peak throughput at around 19100 transactions per second (tps). The median round-trip latency falls in the range from 22ms to 120ms in both strong and weak coupling methods. The two lines almost overlap with a negligible difference. As I increase the number of `goroutines` per client from 600 to 1000, the throughput does not significantly improve but the median latency increases from 70ms to 120ms. In the second experiment with 5 shards (Figure 3.8b), the strong coupling method has a peak throughput at 29600 tps and the median round-trip latency falls in the range from 22 to 135ms. The strong coupling method outperforms the weak coupling method when the number of `goroutines` per client is more than 400. The weak coupling system reaches the peak throughput at 26700 tps, and the median round-trip latency falls in the range from 22ms to 148ms. In the weak coupling method, increasing the number of `goroutines` per client from 600 to 800 does not significantly improve the throughput but the median latency increases.



(a) 3 shards

(b) 5 shards

Each dot, respectively from left to right, represents the latency and throughput when each client has 100, 200, 400, 600, 800, and 1000 goroutines.

Figure 3.9: Performance of strong and weak temporal coupling methods in simulated environment with a 10ms round trip simulated latency

I repeat the same experiment and use the `tc` command to add 10ms round-trip latency to the system to simulate geographical distribution. The result is shown in Figure 3.9. With the added latency, the gap between the strong and weak coupling methods is widened and the strong coupling method outperforms in both cases with 3 shards and 5 shards. EPaxos commits commands in 1 round-trip time (RTT) in the common case and 2 RTTs when there is a conflict (i.e., when two interfering commands arrive at different acceptors in different orders) [58]. Hence, the simulated latency adds 1 RTT between clients and FE servers, 1 RTT between FE and BE servers to submit and download blocks, and 1-2 RTTs for EPaxos in each shard to order a block. In the weak coupling case, the latency additionally adds 1-2 RTTs for FE servers to order blocks. In Figure 3.9a, the throughput of the weak coupling method is 10% less than that of the strong coupling method, and it has an additional 15 - 20ms latency. In Figure 3.9b, the throughput of the weak coupling method is 5% less than that of the strong coupling method, and it has an additional 10 - 20ms latency.

We observe that both strong and weak temporal coupling systems reach similar peak throughput with a 10-20ms difference in latency. This result is consistent with the simulated latency we add. In both strong and weak coupling methods, blocks are downloaded from the shards in parallel. Meanwhile, in the weak temporal coupling methods, FE servers

Source \ Destination	us-east-1	us-east-2	us-west-1	us-west-2	ca-central-1
us-east-1	N/A	11 ms	61 ms	82 ms	16 ms
us-east-2	11 ms	N/A	50 ms	49 ms	26 ms
us-west-1	62 ms	50 ms	N/A	21 ms	79 ms
us-west-2	81 ms	49 ms	20 ms	N/A	60 ms
ca-central-1	15 ms	26 ms	79 ms	60 ms	N/A

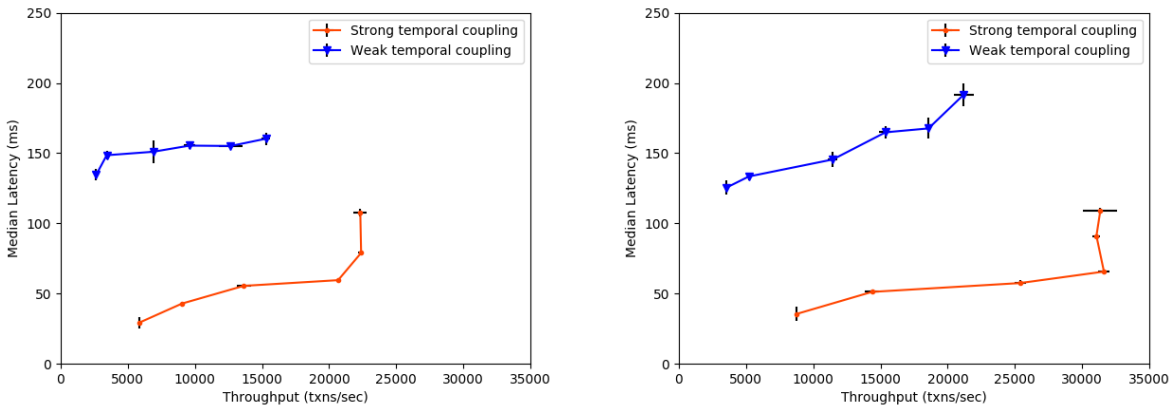
Table 3.1: Round-trip latency across AWS regions

take turns to propose blocks using EPaxos, which takes 1 - 2 additional RTTs [58]. In Section 3.4.2, I bring the experiment to Amazon Elastic Compute Cloud (AWS EC2) across different regions.

3.4.2 On AWS EC2

To experiment with real-world latency, I repeat the same experiment in different Amazon AWS regions. In the case of having 3 shards, I set up the servers in regions us-east-1, us-west-1, and ca-central-1. In the 5-shard case, I set up the servers in regions us-east-1, us-east-2, us-west-1, us-west-2 and ca-central-1. It is an appropriate assumption that an FE server would like to connect to the nearest BE server, and a client would likely send transactions to the nearest FE server. Therefore, in each region, I set up one shard of 3 BE servers, one FE server, and one client program that continuously sends transactions to the FE server in the same region. The round-trip latency is measured across regions, as shown in Table 3.1. The number of cores assigned to each server is similar to those used in Section 3.4.1. Each BE server runs in a t2-micro instance that has 1 vCPU, each FE server runs in a t2-xlarge instance that has 4 vCPUs, and each client program runs in a t2-medium instance that has 2 vCPUs. Similar to what we do in Section 3.4.1, the throughput is controlled indirectly by varying the number of `goroutines` per client. Each dot in Figure 3.10 is corresponding to having 100, 200, 400, 600, 800, and 1000 `goroutines` per client.

Compared to Section 3.4.1, we observe a more significant gap between the strong and weak coupling lines in Figure 3.10. This is due to the higher latency across different AWS regions, compared to the 10ms simulated latency I add in Section 3.4.1. The additional 2



(a) 3 shards

(b) 5 shards

Each dot, respectively from left to right, represents the latency and throughput when each client has 100, 200, 400, 600, 800, and 1000 goroutines.

Figure 3.10: Performance of strong and weak temporal coupling methods on AWS EC2

RTTs for weak-coupling FE servers to order blocks is roughly 100ms, as shown in Table 3.1. In Figure 3.10a, using the method of weak temporal coupling, the system reaches the peak throughput of 13500 tps and the latency falls in the range of 130ms to 180ms. In Figure 3.10b, it reaches the peak throughput of 21200 tps and the latency falls in the range of 125ms to 190ms. Meanwhile, the strong temporal coupling method significantly outperforms the weak temporal coupling method. In Figure 3.10a, the system reaches the peak throughput of 22500 tps and the latency falls in the range of 30ms to 110ms. In Figure 3.10b, it reaches the peak throughput of 27800 tps and the latency falls in the range of 35ms to 110ms. The latency difference between the strong and weak temporal coupling is due to the additional rounds of EPaxos consensus among FE servers in different AWS regions, similar to the difference we observe between Figure 3.8 and 3.9. In the weak coupling method, each FE server takes turns to propose their blocks, which creates two additional RTTs among FE servers [58]. The higher latency, the longer it takes for line 7 in Fig. 3.7 to finish. Thus, the higher latency causes a delay for a transaction `tx` to appear in `H`, and it takes more time for the second while loop in the `receive_tx()` function (line 27 - line 33 in Fig. 3.5) to finish. In addition, there is a limit on the number of concurrent streams that `gRPC` can handle, which means at any given time, there is a limit on the number of `receive_tx()` functions running. When a `receive_tx()` function takes too long to finish, it is possible that another client has to wait before it can kick off a `receive_tx()` function.

In the weak-coupling case, even though each FE server downloads blocks to their channel in parallel, they have to propose blocks in sequence. As a consequence, the throughput decreases, and the latency increases in the weak-coupling case.

3.4.3 Summary

Under the same setup and number of cores, the experiments described in Figures 3.8, 3.9 and 3.10 show a different gap between the strong and weak coupling methods. This is due to the difference in network latency among processes. When there is no simulated network latency (Fig. 3.8), the difference between the two coupling methods is small. The more latency we add to the system, the wider the gap between the two methods (Fig. 3.9 and 3.10). This indicates that the weak coupling method is more susceptible to the increase in latency, which is due to the communication overhead in the consensus layer of weak coupling FE servers. Using EPaxos, the weak-coupling method requires 2 addition RTTs to order blocks among FE servers. However, as discussed in section 3.1, the weak coupling method allows the shards to grow at different rates, while in the strong coupling method, the system stalls if any of the shards fails to respond. Thus, the strong coupling method is desirable when the shards grow at similar rates and allow the servers to be located in a wider location range, while in the weak coupling method, the system continues to work even when one shard fails, but it requires the servers to be in close proximity. Currently, the system depends on EPaxos to batch blocks. In future work, we will try different batching mechanisms and observe whether they improve the performance of the weak-coupling FE servers.

Our system is resilient against malicious clients who want to double-spend transactions. However, a weakness is the inability to handle Byzantine failures among front-end and back-end servers. In future work, we will replace EPaxos with a high-performance BFT consensus protocol. Because we already separate the system into layers (clients, FE servers, and BE servers), we can replace the consensus protocol in the BE servers without interfering with the implementation of the FE servers and clients. The weak-coupling FE servers can also use another BFT consensus protocol. This is because we separate the functionalities of the FE servers into different `goroutines`, where they communicate with each other using concurrent Go `channel`. As shown in Section 3.3.3, we only need to change the `goroutine` for Task (4b) and use a different BFT consensus protocol to order blocks.

Chapter 4

Lightweight Front-End Servers

This chapter presents the use of lightweight front-end servers to help scale up the system. They can validate signatures and help detect double spending but rely on full front-end servers for block verification and interleaving. This chapter is organized as follows. In Section 4.1, I discuss the motivation to use lightweight front-end servers, the use of lightweight nodes in Bitcoin, and the Bloom filter data structure. I describe how I use a vector to check for collision in each Bloom filter, and how collision vectors can be used to check double-spending within each block. In Section 4.2, I explain the change in the system design after adding the lightweight front-end servers and describe the pseudo-code. I also describe a novel use of Bloom filters to help detect double-spent transactions within each block. In Section 4.3, I provide empirical results to show the role of lightweight FE servers in improving the performance, measure the system performance with different sizes of Bloom filters, and show how the ratio of full front-end servers affects the performance of a sharded blockchain.

4.1 Motivation

4.1.1 Simplified Payment Verification

Section 8 of the Bitcoin white paper introduces the concept of Simplified Payment Verification (SPV) [61]. It is a mechanism that helps nodes verify Bitcoin transactions without the need to store the entire blockchain. Instead of having the full copy of the blockchain, an SPV node maintains the chain of block headers. To check if a transaction is in a block, an

SPV node requests a Merkle proof from the sender. If the Merkle proof leads to a Merkle root in a block header, it means the transaction is in that block. Compared to the full nodes in Bitcoin, SPV nodes cannot detect if the transactions in a block are double-spent or in the correct format, and hence they cannot be miners. SPV nodes rely on full nodes and place their trust in the full nodes to verify a block or transaction. The SPV mechanism is lightweight in terms of storage and network bandwidth, which helps more nodes access the blockchain and scale up the system.

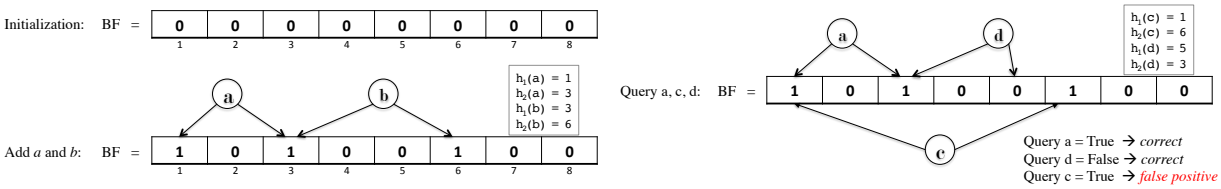
We can find the idea of using lightweight nodes along with full nodes in several sharded blockchain systems. In Elastico [50], OmniLedger [37], and RapidChain [79], only a portion of members participate in block verification and ordering. Thus, to scale up our current system, I add lightweight front-end servers and design a Bloom filter that helps summarize the transactions and check for double-spent transactions within each block.

4.1.2 Bloom Filter

The Bloom filter has been a widely used data structure that quickly tests set membership. It supports membership queries with no false negative error and an acceptable false positive rate. Assume that there are n elements in a set S such that $S \subseteq U$ and U is a universal set. The Bloom filter is a space-efficient probabilistic data structure that represents the n elements in set S using a bit vector of size m, and k hash functions whose domain and range are U. In the initial state, all m bits in the bit vector are set to 0. Let the bit vector be $BF = [0, 0, ..., 0]$. For each element x_i, we insert it into BF by setting $BE[h_j(x_i)] = 1$ for $j \in 1, 2, ..., k$, where h_j is the j^{th} hash function. To test if an element x_i belongs to set S, a Bloom filter returns True if $BE[h_j(x_i)] = 1$ for all $j \in 1, 2, ..., k$; otherwise, it returns False. The insert and query operations are described in Figure 4.1. Thus, the query operation in a Bloom filter has no false negative error, which means if it says an element is not in S, it is certainly not in S. However, it allows for false positive errors. If it says an element is in S, it may not have been inserted to S. In Figure 4.1b, c has not been inserted to BF, but because its hash values match those added to the BF, the Bloom filter falsely detects that c is already inserted.

For a Bloom filter that inserts n elements to a bit vector of size m and has k hash functions, the theoretical false positive rate is $f_r = [1 - (1 - \frac{1}{m})^{nk}]^k$ [57]. Given a desired false positive probability ϵ, the optimal number of bits per element is $\frac{m}{n} = -\frac{\log_2 \epsilon}{\ln 2}$, and the corresponding value of k to is $\frac{m}{n} \ln 2$.

In many applications, the false positive can be capped at a certain threshold, while Bloom filters can save a significant space. For example, in [22], proxies use Bloom filters



(a) Initialization and insertion (b) Query

Figure 4.1: A Bloom filter with $m = 8$, $n = 2$ and $k = 2$.

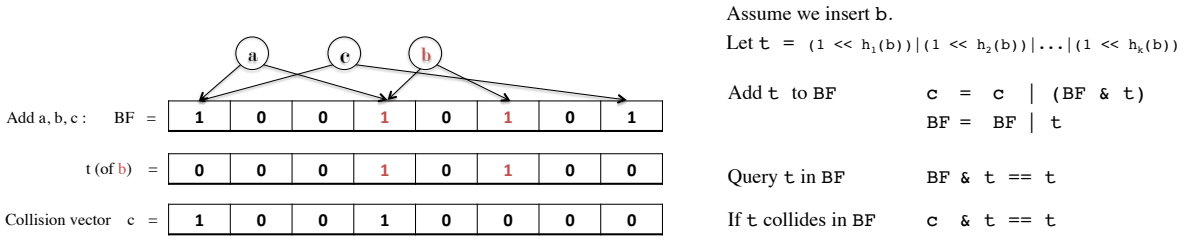


Figure 4.2: A Bloom filter with a collision vector.

as summaries of their Web caches. Instead of sharing the full contents and URL lists of their caches, each proxy builds a Bloom filter from the list of URL's of cached documents and shares the Bloom filter to other proxies. When a proxy P wants to know if another proxy Q has a specific web page wp, proxy P uses its URL to compute the Bloom filter bf that represents wp and checks if proxy Q has a page with Bloom filter bf. If so, proxy P requests the full content of the page from proxy Q. In the case of a false positive error, proxy Q has another webpage wp' that has the same Bloom filter as wp, and it sends wp' instead of wp to P. After proxy P receives the page wp' from proxy Q, it will find out that that page wp is not yet cached in Q. In that case, there is some additional latency that occurs. However, the probability of such an event can be controlled by adjusting the number of hash functions k, the number of bits m, and estimating the number of inserted items n. In this application, the reduction in network traffic significantly outweighs the small probability of a false positive error and its additional latency.

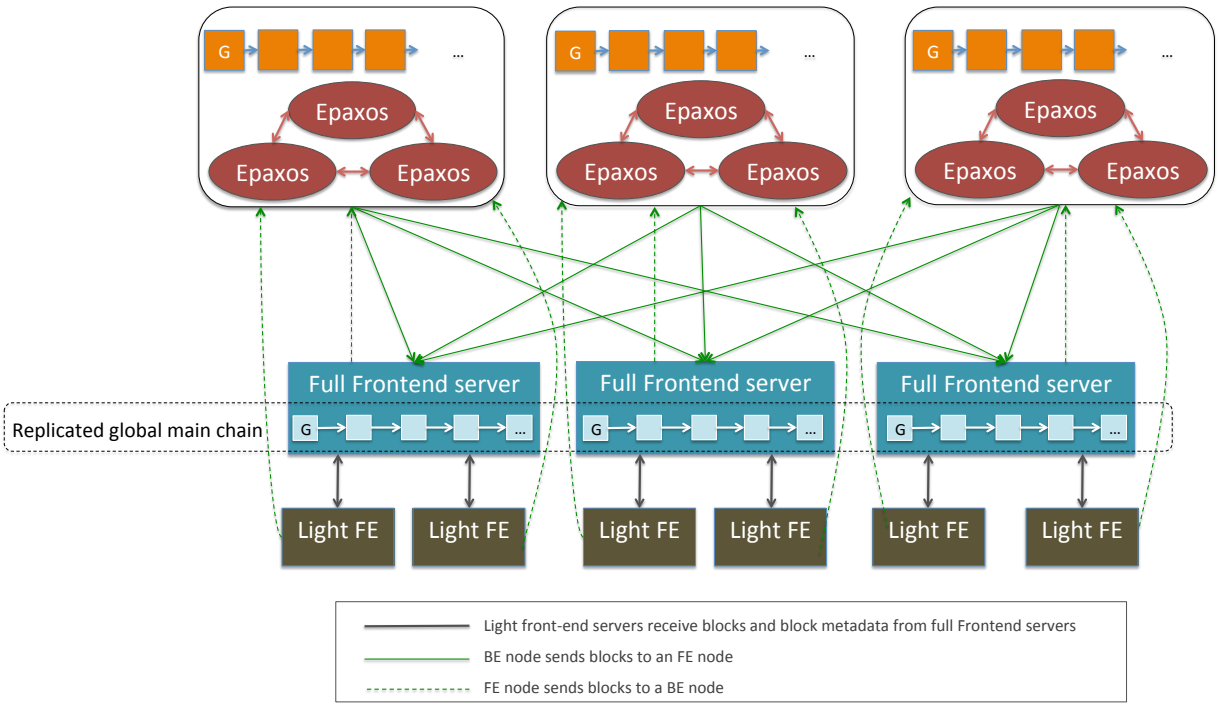
Using a similar idea, for each block, I use a Bloom filter to summarize the content of the transactions. Each block has a Bloom filter that records the hashes of all the transaction inputs (UTX Hashes). For valid transactions, each input can only be spent once, and hence the transaction input hash can act as an identification of a transaction. The Bloom filter can quickly check the membership of a transaction in a block by checking if its input hash is in the Bloom filter. The lightweight front-end servers can store the Bloom filters of all

blocks in the main chain, and only request the full content of blocks that they propose. Hence the lightweight FE servers do not have to store the entire chain. In addition, I use Bloom filters to help detect double-spent transactions within each block. For each block, I first use the global unspent transaction hash-map to check that the inputs have not been spent in an earlier block. Then I check if a transaction is potentially double-spent within each block by checking if its input is potentially inserted more than once to the Bloom filter. I add an m-bit collision vector `c` to check for collision in each Bloom filter, as described in Figure 4.2. For each Bloom filter `bf`, if the i^{th} bit in vector `c` is set to 1, it means that there are at least two inputs inserted to the i^{th} position in `bf`. As UTX hashes are inserted into the Bloom filter, if they collide, there is a potential for double-spending within a block. If a UTX hash does not collide in `c`, it is guaranteed that it is not double-spent within the block. A modified version of the `validateInputs()` function that utilizes Bloom filters to check double-spending within each block is described in Figure 4.4.

4.2 Implementation

The lightweight front-end (LFE) servers submit blocks directly to back-end servers and store the Bloom filters of all blocks of the entire chain. LFE servers do not participate in interleaving blocks and only keep a portion of the main chain. Similar to SPV nodes in Bitcoin, the lightweight FE servers will rely on the full front-end servers for block validation and ordering, but they can request this information from more than one full front-end server to check for correctness. As described in Figure 4.3, full front-end (Full FE) servers interleave blocks and distribute blocks on request to lightweight front-end servers. Full FE servers are similar to the front-end servers described in Chapter 3, with the additional functionality to distribute blocks to lightweight front-end servers. From here, *front-end servers* refer to both full and lightweight front-end servers, and *original front-end servers* refer to the front-end servers described in Chapter 3.

The pseudo-code of full and lightweight front-end servers is described in Figures 4.5 – 4.7. The block and transaction format are the same as described in Section 3.2. Each client has multiple `goroutines` that concurrently send transactions to the nearest FE server, which can be either a full or lightweight FE server. The `receive_tx()` and `append()` functions remain the same as in Figure 3.5. Each FE server verifies the signature of the transaction to make sure that the sender is the owner of the given public address. Then it aggregates transactions into a block and proposes that block to an EPaxos server within a shard. After EPaxos nodes within the shard replicate and save the block, they distribute the blocks to full front-end servers and let them interleave the blocks, either using the



There are client processes sending transactions to both full and lightweight front-end servers, which are omitted in this figure.

Figure 4.3: Adding lightweight front-end servers to the system design

strong temporal coupling or weak temporal coupling method. After full front-end servers verify the transactions in each block and make sure that they are not double spent or fake spent, they generate the block metadata and a Bloom filter bf that summarizes the transactions of that block. The block metadata includes previous block hash, Merkle proof, and timestamp of the block. All the UTX hashes of the transactions in the block are inserted to bf. Then, the full FE servers send the block metadata and the Bloom filter bf to all lightweight front-end servers (Line 25 in Fig. 4.5 and Line 26 in Fig. 4.6) in parallel. Once a lightweight front-end server receives bf, if it matches any Bloom filter of a block that the server has proposed, it sends a request to the full front-end server to get the whole block content (Function `receive_block_metadata()` in Fig. 4.7).

Algorithm 3: Utilizing Bloom filters to check double-spending within each block

```
1  Function receive_tx(stream, client, q, H)
       Input  : A stream of transactions from client
                The global transaction channel q
                The global unspent transaction hashmap H
                The number of bits in Bloom filter m
2      local-q ← ∅ a new local transaction channel
3      while (tx ← a transaction from stream) ≠ EOF do
4          utxHash, pubKey, sig, msg ← UTX hash, public key, signature, and
5                                      signed message in tx
6          go:
7              valid ← utxHash ∈ H && utxHash.receiver owns pubKey
8                      && ecdsa.verify(pubKey, sig, msg)
9              if valid = TRUE then
10                 in ← input UTX hash of tx
11                 tx.loc ← (murmur128 hash of in) % m      // Location of tx in the
                    Bloom filter
12                 add tx to local-q
13                 add tx to q
14             end
15         end
16     end
17...22 Line 27 - 33 in Figure 3.5
18 end
19 Function setBloomfilter(b)
       Input  : A block b
                The number of bits in Bloom filter m
20     ⟨bf, c⟩ ← two vectors, each has m bits set to 0
21     foreach tx ∈ b.Transactions do
22         t ← 1 ≪ tx.loc
23         c ← c | (bf & t)
24         bf ← bf | t
25     end
26     ⟨b.bloomfilter, b.c⟩ ← ⟨bf, c⟩
27 end
   // Modify validateInputs() and use Bloom filters to help check double-spending
      within each block
28 Function validateInputs(H, b)
       Input  : The global unspent transaction hashmap H
                A block b
       Output: If block b is valid and has no fake/double-spent transaction
29     m ← {}    a set of input UTX hashes
30     foreach tx ∈ b do
31         in ← input UTX hash of tx
32         if in ∉ H then return FALSE                  // tx is fake/double-spent
33         if in collides in b.c then
34             if in ∈ m then return FALSE              // tx is fake/double-spent
35             add in to m
36         end
37     end
38     return TRUE
39 end
```

Figure 4.4: Common functions in full and lightweight front-end servers

Algorithm 4: Pseudo-code of strong-coupling full FE servers

```
1  Function main(L, #shards)
       Input  : A list of hosts and port numbers L of backend servers
2      init(L)
3      for i ← 1 to #shards do
4          q_i ← ∅    a channel containing blocks downloaded from shard i
5          go get_blocks(i, L, q_i)
6      end
7      Ψ ← host and port numbers of lightweight FE servers registering using gRPC
8      go interleave(β, H, q_1, q_2, ..., q_n, #shards, Ψ)
9  end
10 Function interleave(β, H, q_1, q_2, ..., q_n, #shards, Ψ)        // Task 4a
       Input  : The global main chain β
                The global unspent transaction hashmap H
                A list of channels of downloaded blocks q_1, q_2, ..., q_n
                Number of BE shards #shards
                The list of lightweight front-end servers Ψ
11     while TRUE do
12         for i ← 1 to #shards do
13             b ← the next block from q_i
14             setBloomfilter(b)
15             if validateInputs(H, b) = FALSE then Go to line 12
16             foreach tx ∈ b do
17                 delete (in ← input UTX hash of tx) from H
18             end
19             b.prevTxHash ← the hash of the last block in β
20             append b to the main chain β
21             go:
22                 foreach tx ∈ b do add tx.hash to H
23             end
24             foreach lfe ∈ Ψ do
25                 go send b.bloomfilter and b.metadata to lfe
26             end
27         end
28     end
29 end
```

Figure 4.5: Strong-coupling full front-end servers pseudo-code

Algorithm 5: Pseudo-code of weak-coupling full FE servers

```
1  Function main(L, id)
       Input  : A list of hosts and port numbers L of backend servers
                The id of the server
2      Π, β, H, s, q ← init(L)
3      q_id ← ∅    a channel containing blocks downloaded from shard id
4      go get_blocks(id, L, q_id)
5      Γ ← the list of other FE servers connected using EPaxos
6      Ψ ← host and port numbers of lightweight FE servers registering using gRPC
7      while TRUE do
8          go:
9              b ← the next block in q_id
10             setBloomfilter(b)
11             propose b to Γ                          // Task (4b): propose a block
12         end
13     end
14 end
15 Function execute_proposal(b, β, H, Ψ)     // Task (4b): process a proposal
       Input  : A proposed block b
                The global main chain β
                The global unspent transaction hashmap H
                The list of lightweight front-end servers Ψ
16...24    Line 11 - 19 in Figure 3.7
25     foreach lfe ∈ Ψ do
26         go send b.bloomfilter and b.metadata to lfe
27     end
28 end
```

Figure 4.6: Weak-coupling full front-end servers pseudo-code

4.3 Evaluation

This section presents the evaluation of the blockchain system with lightweight FE servers and Bloom filters. Section 4.3.1 shows the difference of interleaving blocks using the strong and weak coupling methods, with and without using lightweight FE servers. In Section 4.3.2, I show the how the sizes of Bloom filters can affect the throughputs. In Section 4.3.3, I explore how the ratio of full and lightweight FE servers affect the throughput of the blockchain system.

Algorithm 6: Pseudo-code of lightweight FE servers

```
1  Function main(L)
       Input  : A list of hosts and port numbers L of backend servers
2      Π, β, H, s, q ← init(L)
3      bfSet ← a set of bloomfilters
4      go receive_block_metadata()
5  end
6  Function append(q, Π)                          // Modified version of Task (2)
       Input  : The global transaction channel q
                The global channel Π of connections to backend servers
7      prevTime ← currentTime()
8      while q.size() > 100 or (currentTime() − prevTime) > 100ms do
9...13     Line 38 - 42 in Figure 3.5
14         add b.bloomfilter to bfSet
15     end
16 end
17 Function receive_block_metadata()
       Input  : A stream stream of block metadata and Bloom filter from a full FE
                server
18     while TRUE do
19         bf, bmeta ← Bloom filter and metadata of a block received from stream
20         if bf ∈ bfSet then
21             b ← block with bmeta.bid from a full FE server
22             delete bf from bfSet
23             go:
24                 foreach tx ∈ b do add tx.hash to H
25             end
26         end
27     end
28 end
```

Figure 4.7: Lightweight front-end servers pseudo-code

4.3.1 Adding Lightweight Front-End Servers to the System

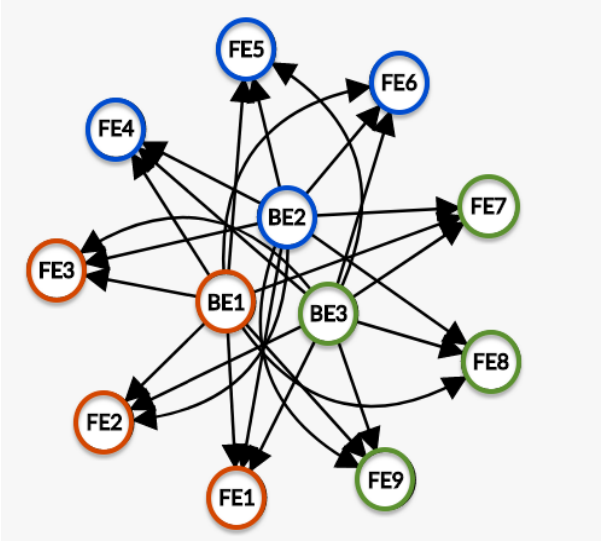
In this section, I describe the experiment on AWS EC2 to test the throughput enhancement by introducing lightweight front-end servers. I set up the shards in 3 different regions: us-east-1, us-west-1, and ca-central-1. In each region, there is one BE shard, three front-end servers, and three clients, which means there are 9 FE servers in total. I compare the

performance of the system with 4 setups as follows. ① All 9 FE servers are the original strong-coupling FE servers using the pseudo-code described in Figure 3.6. ② All 9 FE servers are the original weak-coupling FE servers using the pseudo-code described in Figure 3.7. ③ In each region, there are 2 lightweight front-end servers and 1 full front-end servers, and the 3 full FE servers in 3 regions interleave using the strong-coupling method. The pseudo-code is described in Figures 4.5 and 4.7. And ④ In each region, there are 2 lightweight front-end servers and 1 full front-end servers, and the 3 full FE servers in 3 regions interleave using the weak-coupling method. The pseudo-code is described in Figures 4.6 and 4.7. Each setup has a different way of distributing blocks, as described in Figure 4.8.

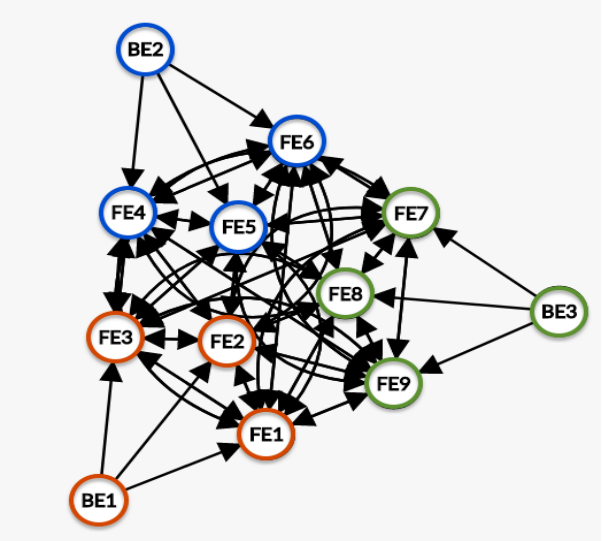
A full FE server connects to the nearest BE server, a lightweight FE server receives blocks from the nearest full FE server, and each client sends transactions to a full/lightweight FE server in the same AWS region. Each BE shard runs in a t2-micro instance that has 1 vCPU, each FE server runs in a t2-xlarge instance that has 4 vCPUs, and each client program runs in a t2-medium instance that has 2 vCPUs. Similar to what we do in Sections 3.4.1 and 3.4.2, the throughput is controlled indirectly by varying the number of `goroutines` per client. Each dot in Figure 4.9 is corresponding to having 100, 200, 400, 600, 800, and 1000 `goroutines` per client. The error bar in the figure represents the standard deviation of the latency and throughput.

As I increase the number of FE servers from 3 to 9, the performance gap between the original strong-coupling FE servers and the original weak-coupling FE servers is widened. The original strong-coupling FE servers (the green line) reach the peak throughput at 45000 transactions per second, and as I increase the number of `goroutines` per client from 600 to 1000, the throughput does not improve, but the latency increases. Meanwhile, the original weak-coupling FE servers (the yellow line) reach the peak throughput at 12500 transactions per second, and the latency falls in the range of 150ms to 875ms.

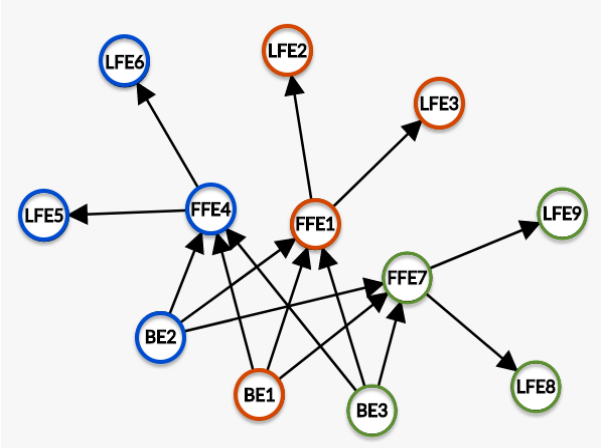
Adding lightweight front-end servers significantly improves the throughput and decreases the latency, as shown in the gap between the red and green lines and between the blue and the yellow lines. When using lightweight FE servers and letting only 3 full FE servers interleave blocks, the system reaches the peak throughput at 56000 transactions per second, and the latency range is 45ms to 110ms. When full FE servers interleave using the weak-coupling method, the system reaches the peak throughput at 36000 transactions per second. Adding lightweight FE servers helps scale up the system, and compared to the original setup, the throughput is not quickly saturated. This improvement is due to the reduction in storage and network bandwidth requirements. On average, each lightweight FE server only saves $\frac{1}{9}$ blocks of the main chain. The communication overhead is reduced as the size of block metadata and Bloom filter is small (48 bytes in total) compared to that



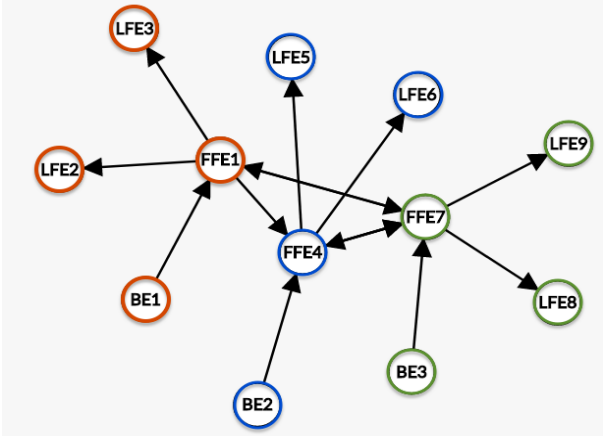
(a) No lightweight FE server. All FE servers interleave blocks using the strong coupling method.



(b) No lightweight FE server. All FE servers interleave blocks using the weak coupling method.



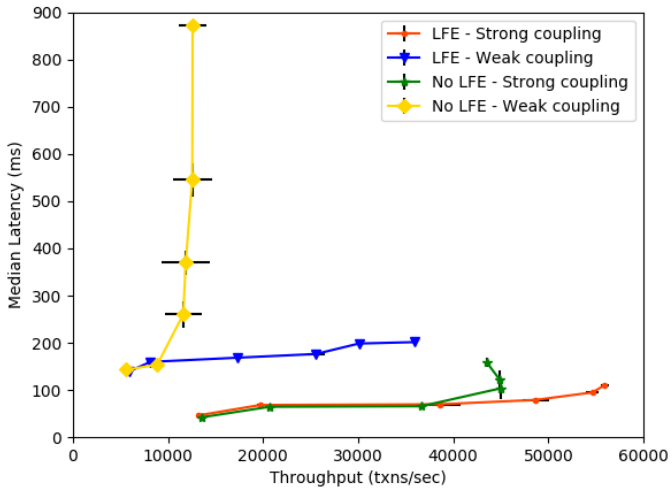
(c) Full FE servers interleave blocks using the strong coupling method and distribute blocks to LFE servers.



(d) Full FE servers interleave blocks using the weak coupling method and distribute blocks to LFE servers.

Processes with the same color are co-located in the same regions.
O us-east-1 O us-west-1 O ca-central-1

Figure 4.8: Four ways to distribute blocks from back-end servers to front-end servers



— Original strong-coupling FE servers - no lightweight FE server (See Fig. 4.8a)
— Original weak-coupling FE servers - no lightweight FE server (See Fig. 4.8b)
— Strong-coupling full FE servers with lightweight FE servers (See Fig. 4.8c)
— Weak-coupling full FE servers with lightweight FE servers (See Fig. Fig. 4.8d)

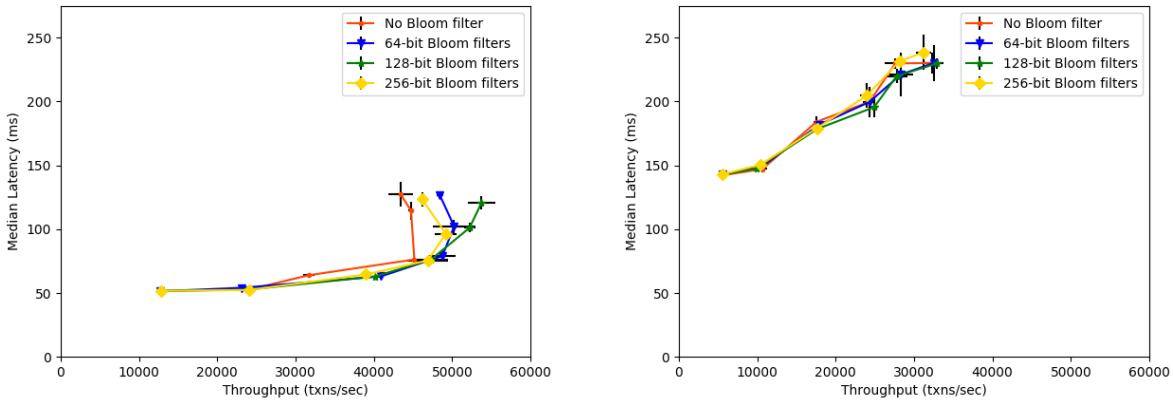
Each dot, respectively from left to right, represents the latency and throughput when each client has 100, 200, 400, 600, 800, and 1000 goroutines.

Figure 4.9: Compare the performances of strong and weak coupling methods, with and without using lightweight front-end servers.

of a block (A block with 1000 transactions is 160-KB long).

4.3.2 Varying Bloom Filter Sizes

In this experiment, I vary the number of bits of the Bloom filters. As shown in Section 4.1.2, the false positive error of a system depends on the number of hash functions k, the number of bits m, and the number of inserted transaction hashes n. I fix $k = 1$ and $n = 50$, and experiment with different values of $m \in \{64, 128, 256\}$-bits. Using AWS EC2 instances in 3 regions, I set up 4 experiments, one experiment does not use Bloom filter to check double-spending within each block (See function `validateInputs()` in Figure 3.5), and 3 experiments that use Bloom filters (See function `validateInputs()` in Figure 4.4). In each of the three experiments, the size of the Bloom filters is $m \in \{64, 128, 256\}$-bits,



(a) Strong coupling with different m

(b) Weak coupling with different m

- Each dot, respectively from left to right, represents the latency and throughput when each client has 100, 200, 400, 600, 800, and 1000 goroutines.
- The red line represents the performance when the system does not use Bloom filters to check for double-spending within each block. The blue, green, and yellow lines represent the performance when using different sizes of Bloom filters $m \in \{64, 128, 256\}$-bits.
- The number of hash functions $k = 1$. The number of transactions per block $n = 50$.

$m =$	64	128	256
False positive rate	0.55	0.32	0.18

Figure 4.10: Performance with and without using Bloom filters to check double-spending within each block.

respectively. The higher m, the lower the false positive rate, but the more bandwidth required to transmit block data. In each region, I set up 1 BE shard that consists of 3 EPaxos servers, 1 full front-end server, 2 lightweight FE servers, and 3 client processes where each client sends transactions to an FE server. Each BE server runs in a t2-micro instance with 1 vCPU, each FE server runs in a t2-xlarge server with 4 vCPUs, and each client runs in a t2-medium server with 2 vCPUs. Each experiment is executed 3 times, each run lasts for 20 seconds, and the throughputs, median latency, and standard deviation errors are shown in Figure 4.10.

As we observe in Figure 4.10b, when the FE servers interleave using the weak coupling method, using Bloom filters to check double-spending within each block does not improve the performance of the system. This is because the consensus layer among FE servers remains the bottleneck, and improving the transaction validation speed does not affect the

overall performance. However, in Figure 4.10a, when the system uses the strong coupling method, we can see that Bloom filters help speed up the system when the number of `goroutine` per client is more than 600. This is because the more `goroutine` per client, the higher the growth rate in each shard. Thus, it is easier to see the benefits of reducing the running time of the `validateInputs()` function. An interesting observation here is that, even though 128-bit Bloom filters have higher false positive rates, they outperform 256-bit Bloom filters. Compared to using 128-bit Bloom filters, using 256-bit Bloom filters doubles the number of required bits but only reduces the false positive rate by 14%. The result suggests that to further improve the performance of strong-coupling FE servers, we should pick the Bloom filter size based on the estimated number of transactions per block.

4.3.3 Ratio of Full Front-End Servers in a System

As discussed in Section 2.3, several sharded blockchain systems hardcode the number of validators and committee sizes in a configuration file before building the blockchain. These numbers are fixed despite the number of active participants in a system. In this section, I discuss the importance of choosing these constants by showing how the ratio of full front-end servers affects the throughputs in our blockchain system. Using AWS EC2 instances, I set up 3 experiments, each has 3 BE shards where each shard consists of 3 EPaxos servers, 36 front-end servers, and 36 client processes where each client sends transactions to an FE server. The servers are separated into 3 regions: us-east-1, us-west-1, and ca-central-1. Each region has 1 BE shard, 12 FE servers, and 12 client processes. Each BE server runs in a t2-micro instance with 1 vCPU, each FE server runs in a t2-xlarge server with 4 vCPUs, and each client runs in a t2-medium server with 2 vCPUs. To calculate the peak throughputs, I adjust the number of `goroutines` per client process from 100, 200, 400, 600, 800 to 1000.

In this experiment, I vary the number of full front-end servers. There are three settings: ① 3 full FE servers and 33 lightweight FE servers; ② 6 full FE servers and 30 lightweight FE servers; and ③ 9 full FE servers and 27 lightweight FE servers. As we observe from Figure 4.11, the fewer full FE servers in the system, the smaller the gap between the performance between the strong and weak temporal coupling methods. In addition, the more full FE servers in the systems, the smaller the peak throughput is. In the strong coupling case, when there are more full FE servers, the BE servers have to send blocks to more FE servers. In the weak coupling case, when there are more full FE servers, there are more nodes participating in the consensus protocol of the blockchain system, which hurts the throughput. This explains why the gap between the strong and weak coupling methods is widened when the number of full FE servers increases. On the other hand, when there

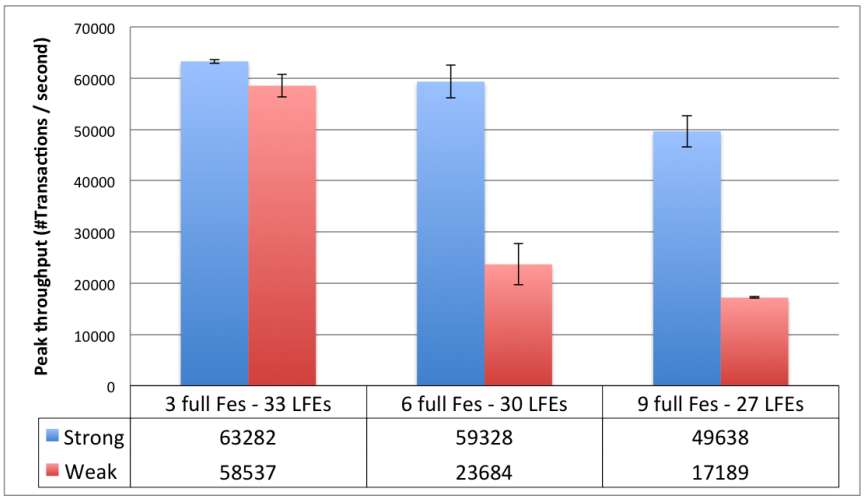


Figure 4.11: Peak throughputs with different number of full front-end servers

are more full FE servers, the system is more fault-tolerant as it can stand more failures of full FE servers. This shows the trade-off between fault-tolerance and performance when adjusting the number of full front-end servers.

Chapter 5

Conclusion and Future Work

This thesis discusses the principles of the sharding concept used in blockchain designs and how current sharding blockchain systems use it to scale out. Section 2.3 discusses the general mechanism used in sharded blockchains, examines different BFT consensus protocols and their approaches to achieve intra-shard and cross-shard consensus.

There are two major approaches to validate cross-shard transactions. One is to build full-mesh connections among nodes, and the other is to build a global root chain. Using the second approach - maintaining a global root chain, we can reduce the data migration overhead when handling cross-shard transactions, replicate the total order of blocks, and maintain security when one shard is controlled by an adversary. This approach raises the question of how we can maintain a global chain of blocks from different shards. In Chapters 3 and 4, I examine two methods to interleave blocks and discuss the role of lightweight servers in a blockchain system. The experimental results in Chapter 3 suggest that when there is negligible latency among servers, both strong and weak temporal coupling methods perform similarly in terms of throughput and end-to-end latency. The weak temporal coupling method can tolerate the crash failures of back-end shards and does not stall if the shards grow at different speeds. However, it requires an additional layer of consensus among front-end servers and hence is more susceptible to the increase in latency among servers. While the strong temporal coupling method requires the shards to grow at a similar rate and cannot tolerate shard failures, using this method to interleave blocks provides a higher throughput compared to using the weak temporal coupling method.

The experimental results in Chapter 4 suggest that adding lightweight front-end servers helps scale out the system. Lightweight front-end servers can help parallelize signatures verification of transactions in blocks and rely on validators - a portion of front-end servers

referred to as full front-end servers - to validate and order blocks. Similar ideas are discussed in Section 2.3, where only a portion of members participate in block verification and ordering. In addition, I propose a novel usage of Bloom filters and collision vectors to summarize of contents of transactions and help check double-spending transactions within each block. Bloom filter sizes are much smaller than the whole content of blocks, which helps save bandwidth and reduce the communication overhead between full and lightweight front-end servers. I also show that the number of bits in Bloom filters can impact the performance. To further improve the throughput, we should pick the number of bits of Bloom filters based on the estimated number of transactions per block.

Several sharded blockchain systems hardcode the number of validators and committee sizes in a configuration file before building the blockchain. Section 4.3.3 shows that the ratio of full front-end servers over lightweight front-end servers in a blockchain system can impact the peak throughput. The more full front-end servers, the more available nodes that can interleave blocks, and the lower the throughput is. This suggests a future sharded blockchain system considers dynamicity instead of depending on constant parameters, specifically when the number of active participants is not fixed. If it needs to hardcode some constants (such as the number of validators and committee size), they should be carefully measured to maximize the performance of the system.

For future implementations, we are going to investigate the data traffic and bandwidth bottlenecks for the system to evaluate data propagation among processes and further improve the throughput and reduce latency. We will change the consensus protocol and replace EPaxos with a BFT consensus protocol, preventing the system from Byzantine faults.

References

[1] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Efficient synchronous Byzantine consensus. *arXiv preprint arXiv:1704.02397*, 2017.

[2] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1337–1347. IEEE, 2019.

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[4] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 585–602, 2019.

[5] Sikha Bagui and Loi Tang Nguyen. Database sharding: to provide fault tolerance and scalability of big data on the cloud. *International Journal of Cloud Applications and Computing (IJCAC)*, 5(2):36–52, 2015.

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

[7] Blockchain.com. Bitcoin total hash rate. https://www.blockchain.com/charts/hash-rate. Accessed on 2020-12-08.

[8] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.

[9] Danny Bradbury. The problem with bitcoin. *Computer Fraud & Security*, 2013(11):5–8, 2013.

[10] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[11] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

[12] Anamika Chauhan, Om Prakash Malviya, Madhav Verma, and Tejinder Singh Mor. Blockchain and scalability. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 122–128. IEEE, 2018.

[13] Wubing Chen, Zhiying Xu, Shuyu Shi, Yang Zhao, and Jun Zhao. A survey of blockchain applications in different domains. In *Proceedings of the 2018 International Conference on Blockchain Technology and Application*, pages 17–21, 2018.

[14] D. Collins, R. Guerraoui, J. Komatovic, P. Kuznetsov, M. Monti, M. Pavlovic, Y. Pignolet, D. Seredinschi, A. Tonkikh, and A. Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38, 2020.

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[16] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Dbft: Efficient leaderless Byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2018.

[17] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating the red belly blockchain. *arXiv preprint arXiv:1812.11747*, 2018.

[18] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International conference on financial cryptography and data security*, pages 106–125. Springer, 2016.

[19] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. *arXiv preprint arXiv:1505.06895*, 2015.

[20] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[21] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[22] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3):281–293, 2000.

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

[24] Ethereum Foundation. Ethereum 2.0. https://ethereum.org/en/eth2. Accessed on 2020-10-02.

[25] Ethereum Foundation. Ethereum 2.0 - The beacon chain. https://ethereum.org/en/eth2/the-beacon-chain/. Accessed on 2020-10-02.

[26] Litecoin Foundation. Litecoin.org. https://www.litecoin.org/. Accessed on 2020-10-02.

[27] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

[28] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE, 2019.

[29] Yue Hao, Yi Li, Xinghua Dong, Li Fang, and Ping Chen. Performance analysis of consensus algorithm in private blockchain. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 280–285. IEEE, 2018.

[30] Algorand, Inc. Algorand.com. https://www.algorand.com/. Accessed on 2020-10-02.

[31] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[32] Srinivasan Keshav, M. Wojciech Golab, Bernard Wong, Sajjad Rizvi, and Sergey Gorbunov. Rcanopus: Making canopus resilient to failures and Byzantine faults. *arXiv: Distributed, Parallel, and Cluster Computing*, 2018.

[33] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[34] Soohyeong Kim, Yongseok Kwon, and Sunghyun Cho. A survey of scalability solutions on blockchain. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1204–1207. IEEE, 2018.

[35] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

[36] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th {usenix} security symposium ({usenix} security 16)*, pages 279–296, 2016.

[37] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[38] Sinan Küfeoğlu and Mahmut Özkuran. Bitcoin mining: A global review of energy and power demand. *Energy Research & Social Science*, 58:101273, 2019.

[39] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, 1998.

[40] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 2001.

[41] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

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

[43] Leslie B Lamport. Generalized paxos, April 13 2010. US Patent 7,698,465.

[44] Wenting Li, Sébastien Andreina, Jens-Matthias Bohli, and Ghassan Karame. Securing proof-of-stake blockchain protocols. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 297–315. Springer, 2017.

[45] Google, LLC. golang.org/. https://golang.org/. Accessed on 2020-10-02.

[46] Google, LLC. golang.org/x/net. https://godoc.org/golang.org/x/net. Accessed on 2020-10-02.

[47] Google, LLC. grpc.io. https://grpc.io/. Accessed on 2020-10-02.

[48] Google, LLC. Protocol buffers documentation. https://developers.google.com/protocol-buffers. Accessed on 2020-10-02.

[49] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials*, 21(2):1912–1949, 2018.

[50] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30, 2016.

[51] Chunyu Mao, Anh-Duong Nguyen, and Wojciech Golab. Performance and fault tolerance trade-offs in sharded permissioned blockchains. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–3. IEEE, 2020.

[52] Parisa Jalili Marandi, Carlos Eduardo Bezerra, and Fernando Pedone. Rethinking state-machine replication for parallelism. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 368–377. IEEE, 2014.

[53] Thomas McGhin, Kim-Kwang Raymond Choo, Charles Zhechao Liu, and Debiao He. Blockchain in healthcare applications: Research challenges and opportunities. *Journal of Network and Computer Applications*, 135:62–75, 2019.

[54] Esther Mengelkamp, Benedikt Notheisen, Carolin Beer, David Dauer, and Christof Weinhardt. A blockchain-based smart grid: towards sustainable local energy markets. *Computer Science-Research and Development*, 33(1-2):207–214, 2018.

[55] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.

[56] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2016.

[57] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM transactions on networking*, 10(5):604–612, 2002.

[58] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, 2013.

[59] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Epaxos code base. *https://github.com/efficient/epaxos*, 2015.

[60] Satoshi Nakamoto. Bitcoin. https://github.com/bitcoin/bitcoin, 2020.

[61] Satoshi Nakamoto and A Bitcoin. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin.org/bitcoin.pdf*, 2008.

[62] Lan N Nguyen, Truc DT Nguyen, Thang N Dinh, and My T Thai. Optchain: optimal transactions placement for scalable blockchain sharding. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 525–535. IEEE, 2019.

[63] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[64] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong. Performance analysis of private blockchain platforms in varying workloads. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, 2017.

[65] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A scalable and massively parallel consensus protocol. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 426–438, 2017.

[66] Sara Saberi, Mahtab Kouhizadeh, Joseph Sarkis, and Lejia Shen. Blockchain technology and its relationships to sustainable supply chain management. *International Journal of Production Research*, 57(7):2117–2135, 2019.

[67] David Schwartz, Noah Youngs, Arthur Britto, et al. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5(8), 2014.

[68] Siddhartha Sen and Michael J Freedman. Commensal cuckoo: Secure group partitioning for large-scale services. *ACM SIGOPS Operating Systems Review*, 46(1):33–39, 2012.

[69] David Shrier, Weige Wu, and Alex Pentland. Blockchain & infrastructure (identity, data security). *Massachusetts Institute of Technology-Connection Science*, 1(3):1–19, 2016.

[70] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. Ieee, 2017.

[71] Florian Tschorsch and Björn Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Communications Surveys & Tutorials*, 18(3):2084–2123, 2016.

[72] Robbert van Renesse. BoscoChain: Keeping Byzantine consensus for bockchains simple and flexible. *Presentation at University of Waterloo*, November 2017.

[73] Marie Vasek, Micah Thornton, and Tyler Moore. Empirical analysis of denial-of-service attacks in the bitcoin ecosystem. In *International conference on financial cryptography and data security*, pages 57–71. Springer, 2014.

[74] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In Jan Camenisch and Doğan Kesdoğan, editors, *Open Problems in Network Security*, pages 112–125, Cham, 2016. Springer International Publishing.

[75] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. Sok: Sharding on blockchain. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, AFT '19, page 41–61, New York, NY, USA, 2019. Association for Computing Machinery.

[76] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[77] Lei Yang, Vivek Bagaria, Gerui Wang, Mohammad Alizadeh, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Scaling bitcoin by 10,000 x. *arXiv preprint arXiv:1909.11261*, 2019.

[78] G. Yu, X. Wang, K. Yu, W. Ni, J. A. Zhang, and R. P. Liu. Survey: Sharding in blockchains. *IEEE Access*, 8:14155–14181, 2020.

[79] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948, 2018.

[80] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE international congress on big data (BigData congress)*, pages 557–564. IEEE, 2017.

[81] Qiheng Zhou, Huawei Huang, Zibin Zheng, and Jing Bian. Solutions to scalability of blockchain: A survey. *IEEE Access*, 8:16440–16455, 2020.

APPENDICES

Appendix A

FLP Impossibility

The famous short paper *Impossibility of Distributed Consensus with One Faulty Process* [23], often referred to as the FLP impossibility proof, defines the upper bound of the possible goals with distributed processes in an asynchronous environment.

In short, the FLP impossibility proof shows that in an asynchronous distributed system, if there is at least one process crash, no deterministic algorithm can solve the consensus problem. The paper formally defines the system model as follows. Assume there are $N > 2$ processes that communicate with each other. For simplicity, the paper assumes that these processes need to agree on one of the values $\{0, 1\}$. They send messages of form (p, m) where p is the receiving process, and m is the contents of the message. There are two supported operations on messages: $send(p, m)$ places the message (p, m) in the message buffer, and $receive(p)$ returns either m from a message (p, m) sent to p, or a null message θ if the message buffer is empty. The messages can come in any order and may be delayed arbitrarily, but not lost. By calling $receive(p)$ indefinitely, a process will eventually receive all the messages. A *configuration* is defined as the internal state of all processes and their message buffers. An *event* or *step* happens when a process p performs $receive(p)$ and transitions from the current state to another. In the paper, a *schedule* is defined as execution with a sequence of events, and such sequence is referred to as a *run*. A run is *admissible* when there is at most one faulty process and every message is delivered eventually. A run is *deciding* when a process eventually decides on a proposed value according to the consensus properties. A consensus protocol is *totally correct* if all admissible runs are deciding runs.

Let P be a totally correct consensus protocol despite one faulty process. A configuration C is bivalent if there are two decision values of configurations reachable from C, and i-valent

if it results in only one value i. The paper states three lemmas as follows:

> LEMMA 1. Suppose that from some configuration C, the schedules σ_1, σ_2 lead to configurations C_1, C_2, respectively. If the sets of processes taking steps in σ_1 and σ_2, respectively, are disjoint, then σ_2 can be applied to C_1 and σ_1 can be applied to C_2, and both lead to the same configuration C.

The first lemma expresses a commutativity property of schedules. Since σ_1 and σ_2 do not interact, the result above follows from the system definition.

> LEMMA 2. P has a bivalent initial configuration.

Suppose that the opposite is true. Since all results must be possible, some initial configurations result in a '0' being decided and some result in an '1' being decided. We can order all possible initial configurations in a chain where two adjacent configurations only differ in the starting value of one process p. Along this chain, there must exist two adjacent configurations where one results in '0' and the other results in '1'. We name them C_0 and C_1, respectively. If p fails, it neither sends nor receives any messages and its initial value cannot be seen by any other process. C_0 must still decide on 0, and C_1 must sti ll decide on 1, because the consensus protocol P is totally correct despite one faulty process. As a consequence, C_0 and C_1 decide on different values although they take the exactly same sequence of steps, which contradicts the assumption that the result is predetermined by the initial configurations. Hence, there exists some initial configuration C in which the decision is not predetermined but a result of the order of received messages and whether there is any crash.

> LEMMA 3. Let C be a bivalent configuration of P, and let $e = (p, m)$ be an event that is applicable to C. Let $\mathcal{C}$ be the set of configurations reachable from C without applying e, and let $\mathcal{D} = e(\mathcal{C}) = \{e(E)|E \in \mathcal{C}$ and e is applicable to E$\}$. Then, $\mathcal{D}$ contains a bivalent configuration.

Suppose that $\mathcal{D}$ does not contain a bivalent configuration. First, the authors show that $\mathcal{D}$ must contain both 0- and 1-valent configurations if it has no bivalent configuration. Since C is bivalent, there must exist at least one 0-valent configuration and one 1-valent configuration reachable from C. Let them be E_0 and E_1, respectively. If $E_0 \notin \mathcal{C}$, let $F_0 = e(E_0)$, which belongs to $\mathcal{D}$. Otherwise, there must exist a configuration F_0 that precedes E_0 and belongs to $\mathcal{D}$. Since one of E_0 and F_0 is reachable from each other and F_0

is univalent since it is in $\mathcal{D}$, F_0 must be 0-valent. The same argument goes to E_1 and F_1, hence $\mathcal{D}$ contains both 0-valent and 1-valent configurations. Applying a similar argument from lemma 2, $\mathcal{D}$ must contain a bivalent configuration. From lemma 2 and 3, any process must start from a bivalent initial configuration C_0. From C_0 we can reach to another bivalent configuration, and this may continue forever if a message is delayed arbitrarily to a process. Not all admissible runs are deciding, hence the protocol is not totally correct.