Scaling Permissioned Blockchains via Sharding

by

Chunyu Mao

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Chen Feng Associate Professor, School of Engineering University of British Columbia
Supervisor(s):	Wojciech Golab Associate Professor, Dept. of Electrical and Computer Engineering University of Waterloo
Internal Member:	Derek Rayside Associate Professor, Dept. of Electrical and Computer Engineering University of Waterloo
Internal Member:	Seyed Majid Zahedi Assistant Professor, Dept. of Electrical and Computer Engineering University of Waterloo
Internal-External Member:	Bernard Wong Associate Professor, David R. Cheriton School of Computer Science University of Waterloo

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

This thesis includes first-authored and peer-reviewed materials that have appeared in conference proceedings published by the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM).

ACM's policy on the reuse of published materials in a dissertation is as follows:

> *"Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included."*

The following list serves as a declaration of the Versions of Record for works included in this dissertation:

Portions of Chapter 1, 2, and 3:

Chunyu Mao, Anh-Duong Nguyen, and Wojciech Golab, "Performance and Fault Tolerance Trade-offs in Sharded Permissioned Blockchains," IEEE International Conference on Blockchain and Cryptocurrency (ICBC), 2020, pp.1-3, doi:10.1109/ICBC48266.2020.9169425.

Chunyu Mao, Anh-Duong Nguyen, and Wojciech Golab, "Performance and Fault Tolerance Trade-offs in Sharded Permissioned Blockchains," 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), 2021, pp. 185-192, doi: 10.1109/BRAINS52497.2021.9569789.

Portions of Chapter 1, 2, and 4:

Chunyu Mao, Wojciech Golab, and Bernard Wong, "Antipaxos: Taking Interactive Consistency to the Next Level," 23rd International Conference on Distributed Computing and Networking (ICDCN), 2022, pp. 128–137, doi: 10.1145/3491003.3491012.

Portions of Chapter 1, 2, and 5:

Chunyu Mao and Wojciech Golab, "Sharding Techniques in the Era of Blockchain," 40th International Symposium on Reliable Distributed Systems (SRDS), 2021, pp. 343-344, doi: 10.1109/SRDS53918.2021.00041.

Chunyu Mao, Wojciech Golab, "GeoChain: A Locality-Based Sharding Protocol for Permissioned Blockchains," 24th International Conference on Distributed Computing and Networking (ICDCN), 2023, pp. 70-79, doi: 10.1145/3571306.3571392.

Abstract

This thesis considers scaling permissioned blockchains via sharding techniques. Traditional distributed systems, such as those used in banking and real estate, require a trusted third party to operate and maintain them, which is highly dependent on the reliability of the operator. Since Bitcoin was introduced by Nakamoto in 2008, blockchain technology has been considered a promising solution to the trust issue raised by the traditional centralized approach. Blockchain is now used by most cryptocurrencies and has meaningful applications in other areas, such as logistics and supply chain management. However, scalability remains a major limitation. Various techniques are being investigated to tackle the scalability issue. Sharding is an intuitive approach to improving the scalability of blockchain systems.

First of all, two techniques are examined for interleaving the shards of permissioned blockchains, which are referred to as strong temporal coupling and weak temporal coupling. The analysis and experiment results show that strong coupling loses performance when different shards grow unevenly, but outperforms weak coupling in a wide-area environment due to its inherent efficiency. Weak coupling, in contrast, deals naturally with load imbalance across shards and in fact tolerates shard failures without any additional effort, but loses performance when running on a high-latency network due to the additional coordination performed.

Second, we propose Antipaxos, a leaderless consensus protocol that reaches agreement on multiple proposals with a fast path solution in the failure-free case and falls back on a slow path to handle other cases. A new agreement problem, termed as *k-Interactive Consistency* is formalized first. Then, two algorithms to solve this problem are proposed under the crash failure model and Byzantine failure model, respectively. We prove the safety and liveness of the proposed algorithms and present an experimental evaluation of their performance in the Amazon cloud. Both the crash-tolerant and Byzantine-tolerant designs reach agreement on n batches of proposals with $\Theta(n^2)$ messages. This leads to the linear complexity of each batch in one consensus cycle, rather than a single batch of proposals per cycle in conventional solutions. The experiments show that our algorithms achieve not only lower execution latency but also higher peak throughput in the failure-free case when deployed in a geo-distributed environment.

Lastly, we introduce a full sharding protocol, Geochain, for permissioned blockchains. The transaction latency is minimized by clustering participants using their geographical properties, locality. In addition, the locality is also being used to decide the transaction placement which results in a low ratio of cross-shard transactions for applications, such

as everyday banking, retail payments, and electric vehicle charging. We also propose a client-driven efficient mechanism to handle cross-shard transactions and present analysis. This enables clients to manage their assets across different shards directly. A prototype is implemented on top of Hyperlegder Fabric v2.3 and evaluated on Amazon EC2. The experiments show that our protocol doubles the peak throughput, even with a high ratio of cross-shard transactions, while minimizing the transaction latency.

Acknowledgements

I would like to give my great appreciation to my supervisor, Prof. Wojciech Golab, for his patient guidance and motivation throughout my Ph.D. studies at the University of Waterloo. I would not have been able to complete this thesis without his directive advice. I would also like to thank all the members of my examination committee, Prof. Seyed Majid Zahedi (internal member), Prof. Derek Rayside (internal member), Prof. Bernard Wong (internal-external member), and Prof. Chen Feng (external examiner), for their insightful feedback during the entire course of my study. I would also thank the members of my background exam committee, Prof. Mahesh Tripunitara, and Prof. Paul Ward, for their valuable feedback.

I would also be grateful to colleagues in distributed algorithms and systems lab. It is a great opportunity to work with everyone in the lab. I want to say thank you to Anh-Duong Nguyen, Diego Cepeda, Hao Tan, Sakib Chowdhury, Chien-Chen (Joseph) Chen for their collaboration and support of my research.

I would like to express my special appreciation to my uncle Dr. Yingming Zhao and my aunt Li Wang for their support and inspiration. I am gratefully indebted to their advice on my life. Also, I must express my profound gratitude to my parents for their support and continuous encouragement throughout my years of study.

At last, I want to thank Ripple, Huawei, and the Faculty of Engineering for sponsoring this research. It is also my great honour to study at the University of Waterloo.

Dedication

To my parents

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed systems are widely used in the modern world, such as the Internet, cloud computing, and distributed databases. In computer science, a distributed system is a system that consists of a set of computing components working coherently as a single system to its users. The components are located in different places, running concurrently and coordinated by passing messages [101]. Traditional distributed systems, such as those used in banking and real estate, are essentially centralized, which requires a trusted third party to operate and maintain the system. However, if the system operator is hacked, then the system would become unreliable. Since Bitcoin was introduced by Nakamoto in 2008 [77], blockchain technology has been considered a promising solution to the trust issue raised by the traditional centralized trust approach. Due to its decentralized essence, a trusted third party is no longer needed since every peer in the system can monitor and verify the validity of each transaction. The distinguishing feature of a blockchain, as compared to a conventional decentralized database, is the use of cryptographic hashes to link together batches of transactions (called blocks) in a tamper-resistant manner. This fundamental security feature is vital in cryptocurrencies as it prevents double-spending, and has meaningful applications in other areas of commerce including logistics and supply chain management.

1.1 Blockchain

Blockchain is a decentralized distributed system where each peer maintains the same chain of blocks, which is also called a ledger. Each block contains multiple transactions and links to its previous block using hashing. As a result, if someone wants to change a block after

it is committed, all the blocks after that block have to be updated accordingly and agreed upon by all the participants. Due to this mechanism, blockchain is considered a promising technology for solving the trust issue of traditional distributed systems. Blockchain can be categorized into permissionless blockchain and permissioned blockchain. In brief, permissionless blockchain means that the blockchain system does not have any access control, anyone can join and contribute to the system operation and maintenance, such as Bitcoin [77], Ethereum [18]. A permissionless blockchain is also termed as a public blockchain. In contrast, only a set of approved participants can access the system in a permissioned blockchain, such as Red Belly [23, 22], Hyperledger Fabric [5]. Permissioned blockchain can be further categorized into private blockchain and consortium blockchain. A private blockchain is fully controlled by a party and has limited access permission. Consortium blockchain, instead of a party full control, has multiple parties govern the system corporately.

A technical survey of Bitcoin technology is given by Tschorsch and Scheuermann [99]. They discuss different Proof of X schemes as well as alternative currencies to Bitcoin, which are termed as altcoins. A comprehensive introduction to Bitcoin and cryptocurrency from a technical point of view is given in [78] which explains the details of how Bitcoin works and presents several alter technologies. Crosby [24] gives an introduction to blockchain technology. It briefly describes the history and how blockchain works. It also summarizes the risks for adoption as well as corporate funding and interest. Gosele and Sandner [37] introduce various criteria for the evaluation of blockchain suitability. It not only analyzes several mobility sector use cases with the introduced criteria but also presents challenges of blockchain technology.

1.2 Blockchain Applications

Because of the advantages of blockchain in solving the trust issue, it is considered to be a promising technology in the coming decades. More and more blockchain-based applications are being developed that spread in different areas. The most famous one is Bitcoin [77] in cryptocurrency, which opens the door to blockchain development. But there are some other excellent applications.

The internet blockchain [40] proposes to secure the Internet Border Gateway Protocol (BGP) and Domain Name System (DNS) infrastructure with the help of blockchain technology. It introduces a tamper-proof internet resource management without depending on Public Key Infrastructure (PKI) or the root of trust. Storj [106] proposes a blockchain-based decentralized cloud storage solution that enables a user to transfer and share data

without reliance on a trusted third party. ByzCoin [50] is a scalable algorithm that uses Byzantine Fault Tolerance (BFT) protocol and collected signing to increase the security and performance of the Bitcoin system. It claimed that it can be applied in any blockchain-based system. The Tangle [87] is a cryptocurrency for the Internet of Things. The main difference of Tangle in comparison with other cryptocurrency systems is that it employs a directed acyclic graph instead of a chain of blocks for storing transactions. Pegged Sidechains [7] is a new technology that enables transfers between Bitcoins and other blockchain systems. It can make current cryptocurrency more easily to interoperate. bloXroute [49] is a blockchain distribution network (BDN) that utilizes a new set of servers to provide high-speed transactions and block propagation. It operates underneath the blockchain at the network layer and can boost transactions and blocks propagation speed. The lightning network [85] is an off-chain application that aims to process Bitcoin micro-payment. It can scale Bitcoin. xCurrent [86] is a blockchain-based software developed by Ripple for cross-border payment. It can confirm the transaction in a very short time compared to the present traditional methods. Smart Grid [72] presents a proof-of-concept of the local energy market trading model which is based on a private blockchain. It also evaluates the economic effects by showing the prospect of cost reduction.

In brief, blockchain attracts more and more industrial interests and will have a huge impact on the future world.

1.3 Motivation

As stated above, blockchains are divided into permissionless blockchains and permissioned blockchains. Permissionless means that the blockchain system does not have any access control, anyone can join and contribute to the system operation and maintenance. To ensure a consistent view, permissionless blockchains are mainly relying on the Proof of Work (PoW) scheme to reach agreement on the sequence of blocks. However, PoW consumes a large amount of energy as every participant has to repeat the same computing process. In addition, the smaller the time interval between every two consecutive blocks, the higher the throughput, and the greater the possibility of a block being revoked. As a result, Bitcoin merely validates 6-7 txns/sec, while Ethereum does 20-30 txns/sec. Proof of Stake (PoS) based blockchain, such as Algorand [35], outperforms Bitcoin by assigning decision-making power to participants based on their stake in the digital asset (e.g., cryptocurrency), and assuming that high-stake players cooperate honestly without explicit economic incentives. Algorand achieves about 1,000 txns/sec. In contrast, only a set of approved participants can access the system in a permissioned blockchain. Permissioned blockchains mostly employ

classical Byzantine fault tolerance (BFT) [57] consensus protocols in ordering the blocks. However, classical BFT protocols, such as PBFT [19], are running within a small quorum, with several to tens of participants. If the quorum size expands to hundreds or thousands, which is normal to have more than a thousand participants in blockchain systems, the performance degrades significantly [98, 19]. Various BFT protocols are proposed that target high performance in blockchain systems, such as SBFT [39], DBFT [21], and MirBFT [98]. Red Belly [23] demonstrates its performance with about 30,000 txns/sec throughput and 3-second latency using 1,000 virtual machines deployed in 14 geo-distributed datacenters. Although there are some works, scalability is still one of the major challenges.

This thesis shows how to scale permissioned blockchains via sharding. Sharding is arguably the dominant technique for improving scalability in distributed systems. It has been applied in various ways to both permissionless [51, 61, 108, 33, 80] and permissioned [23, 22, 100, 25] blockchain systems. Although sharding can yield nearly linear scalability in simple distributed key-value storage systems, its efficacy in scaling blockchains is less well understood as the shards cannot operate completely independently due to tighter security requirements. Notably, blocks of transactions must be hash-chained for tamper resistance, which requires inter-shard coordination.

In addition, recent research investigates leaderless [75, 28, 74, 21, 23] and multi-leader [9, 98] designs to increase scalability, which essentially reaches agreement on multiple proposals from different proposers in every consensus cycle. In the leaderless approach, every process is a proposer. In contrast, only designated leaders can be the proposers in the multi-leader approach. EPaxos [75] allows all replicas to propose requests concurrently. While non-conflicting proposals can be committed on the fast path, concurrent conflicting proposals have to be committed on the slow path and executed with the help of a dependency list. Canopus [89] and RCanopus [48] divide the replicas into multiple groups, where each group proposes a batch of requests in one consensus cycle. The final consensus is achieved by interleaving the requests from each group in round-robin order. Moreover, Mir-BFT [98] uses multiple instances of PBFT [19] to reach agreement on multiple proposals. The major challenge of these approaches is how to handle the failure or asynchrony of one or more proposers while achieving high performance.

Partial sharding and full sharding are the two state-of-the-art approaches that perform blockchain sharding in the modern world. Table 1.1 summarizes the main properties of those two approaches. Partial sharding distributes the transactions into different shards and interleaves the shards to form a consistent final ledger [61]. By applying this approach, the computation and communication, such as signature verification, and smart contract execution, are distributed to different shards, but extra storage and transaction validation are required. In contrast, full sharding records each transaction in some specific shard only

	Partial Sharding	Full Sharding
Properties	Each shard maintains a subset of transactions; Partially sharding computation and communication; The main ledger is required to validate all the transactions;	Each shard maintains a subset of transactions; Sharding computation, communication, and storage; Cross-shard transaction handling techniques are required;
Examples	Red Belly, Elastico	OmniLedger, RapidChain

Table 1.1: Comparison of Partial Sharding and Full Sharding

but requires a special protocol for cross-shard transactions. OmniLedger [51], RapidChain [108], and the trusted-hardware-based sharding protocol (THS) [25] are permissionless full shard protocols while SharPer [4] and Channels [6] are permissioned full sharding protocols.

Thus, my research is mainly focusing on scaling the permissioned blockchain systems via sharding compared to the state-of-the-art technologies.

1.4 Contributions

This thesis includes material from previous publications [67, 68, 66, 64]. First of all, we conduct a comparison of two partial sharding techniques, strong coupling and weak coupling [68]. Although strong coupling does not require a consensus protocol to do the interleaving, which can achieve much better performance in a wide-area deployment, the shard growing rate plays a key role in the strong coupling that may make weak coupling achieve similar peak throughput compared to strong coupling in low latency network. Based on the analysis of strong coupling, a new agreement problem called k-interactive consistency (k-IC) is formalized in both the crash failure model and the Byzantine failure model. Then, a solution to k-IC, Antipaxos [66], is proposed and evaluated. At last, we propose a locality-based full sharding protocol, GeoChain [65], that achieves high scalability of permissioned blockchain systems. The main idea is to assemble the shard by the locality of the processes. We also propose an efficient client-driven protocol to handle cross-shard transactions.

1.4.1 Performance and Fault Tolerance Trade-offs in Sharded Permissioned Blockchains

In chapter 3, we clarify and present state-of-the-art techniques in sharded blockchain interleaving, which we call strong temporal coupling (STC) and weak temporal coupling (WTC) [67, 68]. We analyze the two techniques informally. The analysis predicts that STC and WTC can in principle achieve similar transaction throughput, while WTC experiences higher latency due to an additional layer of coordination. However, if different shards do not grow uniformly, then WTC can outperform STC in terms of throughput. On the other hand, STC can outperform WTC in a geo-distributed deployment as the high latency of network links amplifies coordination overhead. To verify the predictions of the model, a unified software platform is implemented that support different interleaving protocols. The platform uses the efficient EPaxos [75] consensus protocol, which helps us expose performance bottlenecks in other layers of the system. The experimental results show that WTC achieves slightly better scalability than STC in a single datacenter deployment due to non-uniform shard growth. However, STC performs much better than WTC in a geo-distributed setting. Both results corroborate the predictions of our analysis and continue to hold even after we augment STC with a mechanism for tolerating shard failures.

1.4.2 Antipaxos: Taking Interactive Consistency to the Next Level

In chapter 4, we formalize a new agreement problem called k-interactive consistency (k-*IC*), in both the crash failure model and the Byzantine failure model. k-*IC* combines Interactive Consistency [83] and Vector Consensus [29] with a parameterized number of values that suits different synchrony situations in one system. Compared to traditional agreement problems, k-*IC* requires processes to agree on an ordered collection of values instead of a single value in each consensus cycle. We present Antipaxos (AP) [66], a solution to k-*IC*. Conceptually, AP takes the fast path concept to the next level by using a simple and efficient all-to-all broadcast as the primary means to disseminate multiple proposals concurrently, and resorting to a more complex mechanism only as the slow path to deal with failures and asynchrony. More importantly, when a failure or asynchrony causes some processes to perform fast path execution while others do not, AP still ensures a consistent view. AP solves the k-*IC* problem for two distinct k values under different conditions. We establish the safety and liveness properties of AP and conduct experiments in the Amazon

cloud to evaluate the performance. Antipaxos achieves high performance in its fast-path execution compared to state-of-the-art algorithms.

1.4.3 GeoChain: A Locality-Based Sharding Protocol for Permissioned Blockchains

In chapter 5, we review the state-of-the-art sharding protocols and related backgrounds first [64]. Then the design of the GeoChain protocol, a locality-based sharding protocol, is presented. We use the geographical property to not only form the shard but also distribute the transactions, which not only maximizes the performance of the consensus protocols but also leads to a low ratio of cross-shard transactions in some applications. Then, we propose a client-driven efficient cross-shard transaction processing protocol. Safety and liveness are also discussed. In addition, we discuss the ACID properties of the sharded blockchain. The protocol prototype is implemented on top of Hyperledger Fabric [5] v2.3. We conduct experiments in the geo-distributed Amazon cloud to evaluate the performance of the prototype. The results show that GeoChain achieves high throughput with sub-second latency.

Chapter 2

Background

2.1 How a Blockchain works

Blockchain is a distributed ledger where every participant agrees on the order of transactions. The transactions are packed in blocks, each block is connected to the previous block by its hash value. Due to the uniqueness of the hash value of each block, if anyone wants to change any transaction in any intermediate block, it has to change all the subsequent blocks and ask everyone else to accept the new sequence of blocks. As a result, double-spending, one digital asset spend more than once, cannot be made easily, which makes the blockchain secure. Some key techniques are introduced in the following paragraphs in order to help understand the concept of blockchain.

2.1.1 Hash function

First of all, the hash value is a numerical value produced by a hash function. In mathematics, a function is a mapping f from one set X to another set Y, where each element x in X has a mapping of y in Y. It can be expressed as:

$y = f(x)$ where $x \in X$ and $y \in Y$.

A hash function is a function h that can map a larger data set U into a fixed-size data set T. Typically, the size of T is much less than the size of U [20].

One problem with hashing is that multiple keys may have the same hash value, which is called a collision. Since $|U| > |T|$, there are at least two elements that map to the same

slot. A good hash function has to minimize collisions. If we assume that each element of the input set U has the same probability to be hashed to an element of the output set T, where $|U|$ is n and $|T|$ is m, independently of any other element. Then it is expected that there are $\lceil n/m \rceil$ elements that have the same hash. This is a basic assumption called simple uniform hashing, which is used for analyzing the hash function. A hash function that can satisfy the simple uniform hashing is considered a good hash function. However, in most cases, we do not know the distribution of the keys.

Due to the randomness of the input set, designing a good hash function is difficult. But if we know the keys already, then we can carefully design a hash function with excellent worst-case complexity. Perfect hashing is the hashing technique that can achieve $O(1)$ memory accesses of a search in the worst case when implementing a hash table [20].

In cryptography, the hash function should be a one-way function, which means in practice the function is infeasible to invert. In addition, it should be easy to compute. Also, with a given output, it should be extremely hard to derive the input that can result in this output. Moreover, with a given input, it should be infeasible to find another input that can result in the same output. In further, it should be highly unlikely to find two distinct inputs that can result in the same output. A family of cryptographic hash functions was published by the National Institute of Standards and Technology (NIST), called secure hash algorithms (SHA) [71].

A blockchain system uses the hash value of each block to identify the sequence of the blocks. For example, in Bitcoin, the hash value must have a system-defined number of leading zeros. To achieve this, the hash value is computed by double hashing the block header with the SHA256 algorithm. SHA256 is a hashing algorithm of the SHA family that generates a 256-bit hash for any bit-length input data. The block header is a data structure that has 6 fields, which are a version number, the previous block hash, the Merkle [73] root of all the transactions that will be included in this block (Bitcoin employs Merkle tree data structure to verify the integrity of the transactions), a timestamp, a difficulty target, and a 32-bit number called the nonce. In each block cycle, every miner tries to find the nonce that can result in a valid hash value for its proposed block. The miner who finds such a nonce first will be the unique valid proposer for that block cycle and wins some Bitcoins as a reward.

2.1.2 Transaction

In computer science, a transaction is a data unit that consists of a collection of operations [93]. For example, if A wants to transfer $100 to B, then this transaction includes two

operations: debit $100 from A's account and credit $100 to B's account. A transaction in Bitcoin [77] is a transfer of Bitcoins from one to some others which contain the meta-data, inputs, and outputs. A transaction in Ethereum [18] is a transfer of Ether from one account to another account. The consensus on the order of the transactions is achieved by proof of work (PoW). The transactions will be recorded in a block data structure.

In a traditional database system, a transaction is required to satisfy ACID properties [93]. ACID represents atomicity, consistency, isolation, and durability. Atomicity means that all the operations of the transaction must be either executed, or none of them is executed. The transaction cannot be left in a partially executed state even if a failure occurs. Consistency means that if the database starts from a valid state, then after the execution of the transaction, the system will result in a valid state. A valid state is defined by the system. For example, if account A transfers x money to account B, before and after the transaction execution, the total amount of accounts A and B must remain the same. Isolation means that all the transactions will be executed as if only one transaction is being executed at a time. Even if two transactions execute concurrently, one of them will appear in the system either before the execution of another transaction started or after the execution of another transaction is finished. The widely adopted technical definition of this property is serializability [82]. Durability means that if a transaction is being executed successfully, the database will persist the update even if the system fails. ACID fits the transactions of private blockchains. But it does not fit the transactions of Bitcoin-like public blockchains. Bitcoin system can ensure atomicity, consistency, and isolation. But if an adversary has more than half of the computation power, the durability can be broken.

2.1.3 Fault Tolerance

In a system, if it cannot provide some services as its promises, then we say this system fails [101]. An error is a condition that at least one state is incorrect, and this may lead to failure. A fault is the cause of an error. There are generally three classes of faults, which are transient faults, intermittent faults, and permanent faults. Failures can be modelled as crash failure, omission failure, timing failure, response failure, and Byzantine failure.

In a traditional distributed system, due to the existence of a trusted thirty party, only crash failure should be handled. Crash failure means that a process fails by halting, but it is working correctly before the halt. However, because of the decentralized essence, a blockchain system has to deal with Byzantine failures. Byzantine failure means that a process can crash, delay sending messages, or send arbitrary messages, which means the Byzantine process can have arbitrary behaviours. For example, the Bitcoin system may fail if the majority of hash power is controlled by malicious nodes.

2.1.4 CAP

CAP theorem was first proposed by Eric Brewer at PODC 2000 [16] and then proved by Seth Gilbert and Nancy Lynch in 2002 [36]. It states that any networked shared-data system can have at most two out of three of the following properties:

*Consistency(C): the system has a unique updated copy of the data at a time.

*Availability(A): every node in the system must be able to respond eventually.

*Partition tolerance(P): the system will still function when a network partition occurs.

Bitcoin is an AP system. Even if there is a network partition, the system is still available and keeps processing the transactions. However, the miners in different partitions will have a different view of the main chain. Thus the main chain may be different for different partitions which violates the consistency property.

2.1.5 Consensus

The core of blockchain systems is a consensus protocol [8]. Consensus is a fundamental problem in distributed systems [53] where each process has an input value and multiple remote processes must agree on a value as the output. Consensus [53] has been used extensively to implement state machine replication [92], including in several Blockchain systems [5] [35] [23]. The classical consensus problem is that a set of processes agree on a single proposal among multiple proposals, despite the possibility of failure.

Formally, consensus under crash failure has three properties:

Agreement: Every pair of processes never decides different values.

Validity: If a process decides on a value, then the value must be proposed by one of the processes.

Liveness: Every process eventually decides a value.

Consensus under Byzantine failure also has three properties:

Agreement: Every pair of non-faulty processes never decides different values.

Validity: If all non-faulty processes propose the same value, then all non-faulty processes agree on that value.

Liveness: Every non-faulty process eventually decides a value.

If the processes and the communication network are working without failures, the problem can be solved easily. However, a real system cannot be always dependable. FLP impossibility [34] states that consensus cannot be achieved in an asynchronous environment with crash failures, which means that agreement cannot be made among processes if there is no upper bound on the time of process may take to respond to a protocol message. Nevertheless, Ben-Or [10], as well as Bracha and Toueg [15], introduce randomization to solve consensus in the asynchronous environment. In addition, if the asynchronous condition can be changed to partial synchrony, then consensus can be solved [30]. In a blockchain system, to make everyone agree on the same consequence of blocks, a consensus must be achieved among all the participants in each block cycle even if a failure occurs.

Although there exist many consensus protocols, most of them are primarily concerned with handling failures and competing proposals instead of scalability. This is accomplished through complex message passing and processing, which imposes overhead. Notably, the problem is traditionally solved using single-leader quorum-based techniques, such as variations of Lamport's Paxos protocol [55]. In such protocols, the leader plays a central role and limits scalability. Using these protocols to implement a replicated state machine (RSM) is inherently costly as every batch of state transitions has to be proposed by the leader and requires at least one dedicated consensus cycle.

To relieve the bottleneck of the leader, some protocols incorporate a fast path mechanism that improves common-case performance, and a slow path to handle other cases, such as failures. For example, Fast Paxos [56] uses an enlarged quorum to bypass the Paxos leader in the absence of competing proposals, though suffers a performance penalty when it falls back to its slow path. The Byzantine-tolerant Bosco protocol [96] and Hot-Stuff [107] similarly use single-leader schemes. Other performance optimizations include rotating the leader [9], as well as computing decisions hierarchically [3]. Some other works involve different types of hardware optimizations [45, 105, 84]. Zyzzyva [52] additionally offloads processing to clients. Furthermore, batching [91] and pipelining [89, 48, 58] are two general techniques to increase consensus efficiency. However, most of the above techniques still suffer from the limitation of the single-leader approach, and scalability remains a weak point when implementing an RSM.

Safety and liveness are the two properties that are commonly used to evaluate a consensus protocol. Lamport first introduced safety and liveness properties in 1977 to prove the correctness of multiprocess programs [54]. Safety means "bad things will never happen". In other words, if the program operates with the correct input, it will never give incorrect output. Liveness means "good things must happen eventually". Alternatively, if the program operates with the correct input, it will always terminate.

The most famous consensus protocol is Paxos [55], which was proposed by Leslie Lamport. It can guarantee the safety property, but the liveness is conditioned on the proposer of the protocol. However, it is a crash fault-tolerant protocol that can tolerate f crash failure out of $2f + 1$ nodes [15]. In a blockchain system, since there does not exist trusted operator, crash fault tolerance is not enough. Instead, Byzantine failure has to be considered in system design. In present blockchain systems, Proof of X based and traditional Byzantine Fault Tolerant (BFT) based protocols are the main blockchain consensus mechanisms [104]. Most of the permissionless blockchains employ Proof of X as their consensus scheme. While traditional BFT protocol is mainly used in permissioned blockchains. It is because the traditional BFT protocol requires pre-agreement on the set of participants, which fits the permissioned blockchain. But recent research also tries to apply BFT protocols in permissionless blockchains. Both methods will be explained in the next section.

2.2 PoX and Traditional BFT protocols

2.2.1 Proof of X

The most well-known prominent technique in the Proof of X family is proof of work (PoW), which is also called the Nakamoto consensus. PoW was first introduced in the Bitcoin whitepaper [77]. In comparison to the traditional consensus where the processes have to select one from multiple proposals as the decision, PoW is used to decide who can be the transaction proposer in a block cycle. Both Bitcoin [77] and Ethereum [18] employ PoW to build the blockchain. For a better understanding of the protocol, how PoW works in Bitcoin will be explained as an example. But the concept is the same in any PoW-based blockchain system.

The core of PoW is a mathematical puzzle whose difficulty can be tuned precisely, and whose solution leads to an economic reward for the winner. Specifically, every miner in the Bitcoin system tries to compute a hash puzzle in each block cycle. A miner is a server that participates in the PoW consensus. As explained in the section on the hash function, the puzzle is to find a hash value with a system-defined number of leading zeros by computing a nonce, a bit string, together with the transactions and block metadata. Whoever solves this puzzle will broadcast its result as a block to its neighbours, and its neighbours will disseminate the block to their neighbours. As a result, the block will be transmitted to all the participants in the system. Due to the difficulty of solving this puzzle, everyone should decide on the block only according to the first valid block in that block cycle, and append the block to the main chain. The system uses the longest chain rule to decide

the main chain. It assumes that the longest chain in the system will be treated as the main chain. Thus, every miner can propose transactions to commit, but only transactions proposed by the one who wins the computation competition will be decided to commit. If a winner occurs, every other just follows the decision. Thus the decision process of PoW is essentially decentralized. And as an incentive, the winner will be rewarded with some coins for their contribution.

The advantage of PoW is that if an attacker wants to change a block, it has to redo all the PoW on and after that block and then catch up to beat the honest miners. In this sense, as long as the majority of CPU power is controlled by honest miners, PoW can guarantee no double-spending and all transactions are correct. Considering the safety and liveness of PoW, safety is not guaranteed, since if there is enough hashing power, a longer chain can be created to replace the current agreed chain. Thus the block finality, the affirmation that the block will not be revoked, is not guaranteed. But, the longer the time after the block is committed, the lower the probability the block will be revoked. In Bitcoin, this waiting time is about 60 minutes. In contrast, the liveness property can be guaranteed as the system will always produce a new block in roughly the same time interval. The main drawback of PoW is that the computational cost of the puzzle limits scalability. For instance, Bitcoin sets the expected time to solve the puzzle at 10 minutes, while Ethereum sets it to roughly every 12 seconds with a security tradeoff. This means that Bitcoin can only process a few txns/sec, while Ethereum can process around a few tens of txns/sec. Also, to make sure the transaction will not be reverted, clients have to wait for multiple blocks to confirm the transactions. Normally, in Bitcoin, each transaction has to wait for 6 blocks to confirm the validity with a high probability that the transaction cannot be double-spending.

Proof of Stake (PoS) is regarded as an improvement over PoW, where multiple servers make redundant efforts to solve the same puzzle. Compared to PoW, PoS assigns decision-making power to servers based on their stake in the digital asset (e.g., cryptocurrency), and assumes that high-stake players will cooperate honestly without explicit economic incentives. PoS is being considered in the next evolution of Ethereum, and in several other systems including Algorand [35], which can outperform Bitcoin by a factor of more than 100. Even this level of performance, however, is inferior to the VISA credit card system, which can scale to 24,000 txns/sec [102].

Besides PoW and PoS, there are some other proof of X schemes, like proof of ownership, proof of consensus, proof of activity, etc. But those techniques are not sophisticated enough to be widely deployed. Proof of X scheme is mainly employed by permissionless blockchains.

2.2.2 Traditional BFT Protocols

Another approach to solving the consensus of blockchain is traditional BFT protocols. BFT protocols have been long discussed in academia since the PBFT [19] in 1999. However, in a traditional distributed system, where there is a trusted third party, a malicious attack is not considered in the consensus design. Nevertheless, because of the research and development of blockchain technology, BFT protocols are being adopted to solve the consensus of blockchain systems. Hyperledger Fabric [5] is developing its second software version based on the BFT protocol. The core of Tendermint [17] and Red Belly [23] are also BFT protocols.

A BFT protocol is a consensus protocol that can tolerate Byzantine failure. The most famous BFT protocol is PBFT [19]. In each consensus cycle, one of the replicas is selected as the primary replica, all the others are the backup replicas. In normal-case operations, when a primary replica receives a request from a client, it starts a three-phase protocol. In the pre-prepare phase, the primary replica atomically multicast the request with a sequence number as the pre-prepare message to other backup replicas. If a backup replica accepts the pre-prepare message, it enters the prepare phase by broadcasting a prepare message to all other replicas. If a replica (including the primary) accepts enough valid prepare messages, it adds the request to its log and replies to the client, which is the commit phase. When a client receives enough valid replies from different replicas, it can ensure that the request has been committed by the non-faulty replicas.

In blockchain systems, BFT protocol is always coupled with state machine replication (SMR) in achieving consensus. This makes a big difference compared to PoW. The safety property of PoW is conditioned on the 51% computation power, but liveness is guaranteed in any situation. In contrast, the BFT protocols always guarantee safety, but the liveness property is conditioned on the number of Byzantine nodes. The upper bound number of faulty nodes for BFT consensus to ensure the liveness is $(n-1)/3$ in an asynchronous network where n is the total number of nodes in the protocol [15]. This number can be relaxed to $(n-1)/2$ in a synchronous environment [1]. As a result, BFT-based blockchain can ensure security by design without worry about the 51% attack.

Previous BFT-SMRs, like PBFT[19], are mostly used in small-scale systems, with only a handful number of replicas. This is because the traditional distributed system does not require large-scale replication on one hand. On the other hand, the intensive communication complexity of the BFT protocol limits the scalability of the system. This is the same even for crash fault-tolerant protocol, which tends to send fewer messages than BFT protocols, such as Paxos [55] and Raft [81]. Whereas the blockchain system requires large-scale deployment, more than hundreds or thousands of nodes, many researchers are investigating

new BFT protocols that target the blockchain application.

Zyzzyva [52, 38, 95] uses clients to speculate the order of the requests to reduce the cost and simplify the design of BFT SMR. It improves the performance of existing BFT services and achieves performance levels near theoretical lower bounds on both throughput and latency. FastBFT [59] proposes a novel message aggregation technique based on hardware trusted execution environments to reduce message complexity. It is a fast and scalable BFT protocol in comparison to several existing BFT variants. Honey Badger BFT [74] claimed to be the first practical asynchronous BFT protocol with an atomic broadcast protocol to provide optimal asymptotic efficiency in the asynchronous setting. Stellar consensus protocol (SCP) [60] is a federated Byzantine agreement protocol that uses a quorum slice with an intersection to reach consensus. Nomination from different quorum slices may produce multiple candidates, so SCP requires the application layer to supply some method for combining candidates into a single one. SBFT [39] is a new BFT protocol which is an improvement of PBFT for world-scale deployment. It uses collectors to reduce the communication complexity to linear if the messages are cryptographically signed by threshold signature. Also, an optimistic fast path reduces client communication. RCanopus [48] is an innovative BFT protocol that employs a leaf-only tree structure like Canopus [89] to separate the whole sequence of requests into super-leaves as the first layer. Each super-leaf works as a shard that processes part of the requests. On top of the super-leaf, a Byzantine group is constructed as the second layer to counter Byzantine failures with the help of a Byzantine consensus protocol. And then, the requests are ordered as a whole by combining the results of all the Byzantine groups as the top layer.

2.3 Scalability of Present Blockchains

From the previous section, we can see that consensus protocols play an important role in the Blockchain system. But before we go through the analysis, let us define the meaning of scalability. Scalability is one of the most important design goals of distributed systems [53]. In general, it spans three dimensions, size scalability, geographical scalability, and administrative scalability. Size scalability means the system can scale with respect to its size without huge performance reduction. Geographical scalability means the impact of the distance of the users is not noticeable. Administrative scalability means the system can be easily managed by multiple organizations.

Throughput and latency are the two key factors used to evaluate the scalability in most distributed systems. Blockchain uses them as well. Throughput describes the quantity of requests that a system can process in a fixed time slot. It is usually measured as

transactions per second in blockchain systems. Latency is the time needed to process one specific request. In a blockchain system, confirmation latency is mostly used to measure the time interval from the time that the system receives the transaction to the time that the transaction is appended. For example, the throughput of Bitcoin is roughly 6-7 transactions per second, and the latency is about 10 minutes. This performance is regardless of the size of the system. As a result, Bitcoin is not a scalable system due to its low throughput and high latency.

Now, going deep into why scalability is a problem of Blockchain systems. As introduced before, consensus is the core of Blockchain systems. Proof of X and traditional BFT consensus are the two strategies to achieve consensus [104]. The two most mature blockchain systems, Bitcoin and Ethereum, employ proof of work (PoW) associated with the longest chain rule to achieve consensus. The core of PoW is a hash puzzle that should be calculated in every block cycle. The expected time to solve the puzzle is set according to the overall computation power. In most PoW systems, the number of transactions contained in a block is limited by the block size which is a constant. The expected time to solve the puzzle is also configured as a constant regardless of the total computation power. As a result, the throughput and the latency of PoW-based blockchain systems can be formulated as:

$$\text{latency} = \text{time to solve hash puzzle} \tag{2.1}$$

$$\text{throughput} = \text{number of transactions in a block}/\text{latency} \tag{2.2}$$

From equations 2.1 and 2.2, to increase the system performance, either increase the number of transactions in a block or reduce the time to solve the hash puzzle or both. Since PoW based blockchain system is a peer-to-peer system, and each peer only broadcasts to its neighbours, more transactions in a block will result in a larger block which will put pressure on the network and peers. The Minimum recommended system requirements of Bitcoin are 500 MB/day download and 5 GB/day upload. Thus to maintain a good network communication environment, the block size should be limited. Bitcoin set the block size at 1 MB maximum. Another approach is to reduce the expected time to solve the hash puzzle. The time to solve the hash puzzle is bounded by the total computational power of the system, thus the theoretical upper and lower bounds vary according to the system. Bitcoin set the expected time to solve a block to around 10 minutes, while Ethereum set the expected time to around 12 seconds. By doing this, the puzzle can be computed in a shorter time. However, this will increase the probability that more than one peer solves

it at roughly the same time. In Bitcoin and Ethereum, when more than one peer solves a puzzle in the same block cycle at the same time, all of them are treated as valid. The other peers can select any of them as the successful miner in that cycle and append its block to the main chain of their view. This will result in forks for different peers. This is the reason for the longest chain rule where the system defines the main chain as the longest chain among all the forks. Although fork chains exist, only the longest chain will be accepted as the main one. If the forks do not appear frequently, then after several block cycles, the forks will be ignored since honest peers only append the block to the longest chain. However, if forks appear frequently, it will take more block cycles to converge to a single longest chain. In the worst case, all the forks grow at the same rate, then peers cannot distinguish which one is the correct main chain. Thus, reducing the time to solve the hash puzzle is not a straightforward approach considering the longest chain rule.

Because of the poor scalability of PoW, PoS is being proposed to replace PoW. The idea is, instead of using the hash puzzle to decide who can propose a block in a block cycle, a quorum of peers is selected as a committee to represent all the peers. The committee runs a BFT consensus protocol to decide which block can be appended to the main chain. The longest chain rule is no longer a requirement. This can remove the redundant puzzle computation performed by every peer. However, PoS is still not mature enough, Algorand [35] is a developing PoS-based blockchain system. One of the difficulties is how to properly select the committee. Another difficulty is how the committee achieves consensus efficiently under Byzantine failure. Other proof of X schemes are also under investigation.

Proof of X schemes are mainly used for permissionless blockchains because of the open membership, although there are some for permissioned blockchains like proof of authority. The consensus should take the peer joining and leaving into consideration. For permissioned blockchains, since there is access control to the membership of the system, peers cannot join and leave arbitrarily. Thus, traditional BFT consensus protocols are considered to implement the system. Compared to proof of X schemes, traditional BFT protocols have much better performance, high throughput, and low latency. However, scalability is also a major drawback of traditional BFT protocols. Most of the traditional BFT protocols are running within a small quorum, several to tens of nodes. If the quorum size expands to thousands, the performance degrades significantly. In contrast, the performance of PoW is not correlated to the number of peers as long as the network is not saturated. On the other side, traditional BFT protocols assume an upper bound on the ratio of Byzantine peers, $< 1/3$ in an asynchronous environment, $< 1/2$ in a synchronous environment [15]. If the ratio exceeds the bound, consensus cannot be achieved. The PoW-based systems assume that the majority of computation power is controlled by honest peers. Nevertheless, PoW-based systems have thousands of peers. Both Bitcoin and Ethereum have around

4000 peers. However, traditional BFT protocols only have tens of peers which are easy to attack. As a result, researchers are working on designing large-scale BFT protocols that can be applied to blockchain systems.

2.4 Blockchain Benchmarking

To benchmark a blockchain system, the most two frequently used metrics are throughput, measured in transactions per second, and transaction confirmation latency, measured in seconds. These two dimensions are measured across different network and fault tolerance conditions. Due to the success of Bitcoin, the unspent transaction output (UTXO) transaction model is widely used in benchmarking. As the name indicates, the UTXO is a set of outputs of unspent transactions. An unspent transaction is a transaction, like a coin, that can be spent once by its owner. A valid transaction includes some inputs and some outputs. The input of any new transaction should use at least one of the unspent transactions in this set and generate a new unspent transaction as the output. Otherwise, the transaction will not be valid. The UTXO set is derived from the chain of blocks and updated for every new block. Another transaction model is the account-based transaction model. In contrast to the UTXO model, every user in the account-based model has a balance on their account. And a transaction is a transfer of balance from some accounts to some other accounts, like the bank system.

Although there exist lots of different blockchain systems, there does not exist any system that can perform a unified comparison among them. The difficulty is that each system has its own specified transaction data structure and system workflow. Blockbench [27] is the first framework proposed in 2017 to evaluate private blockchains. But it only supports Ethereum [18], Parity [32] and Hyperledger Fabric [5]. Caliper [44] is another tool designed for measuring blockchain performance, led by Linux Foundation. It can be used to test several Hyperledger projects, such as Hyperledger Fabric and Hyperledger Sawtooth, etc. However, most of the blockchain systems are only running their own tests and comparing the transaction throughput and latency reported by other systems.

2.5 Major Blockchains

As stated in Chapter 1, the blockchain system can be categorized into permissionless blockchain and permissioned blockchain. Permission means the authority to participate

in the system. This section will introduce several well-known blockchain systems in each category.

2.5.1 Permissionless Blockchains

Bitcoin [77] is the first well-known permissionless blockchain-based cryptocurrency. It was developed by Satoshi Nakamoto in 2008. An interesting fact is that no one knows whether Satoshi Nakamoto represents a person, a group of people, or an organization. It is a fully decentralized ledger that operates by all the miners in the system. The coin is only generated by the system as an incentive to the peer who wins the competition in each block cycle according to a protocol called Proof of Work (PoW). Each transaction of the coin is stored in a chain of blocks connected by the block hash. And this chain is replicated by every miner. Instead of the traditional account-based ledger, Bitcoin is a transaction-based ledger where each transaction records the sender and receiver and saves this information in the ledger. As a result, if anyone wants to modify a transaction in a block, all the blocks on and after that block have to be changed accordingly. However, due to the computation complexity of PoW, it is almost impossible to do that as long as more than 50% of the computation power is controlled by honest miners. Thus, Bitcoin is treated as a secure system by design. Nevertheless, also because of the computation complexity of PoW, the power consumption to maintain the system is extreme. Bitcoin is estimated to cost about 73 TWh per year on average. Moreover, the system can merely process around 7 txns/sec to ensure security and lower the risk of chain forks. Around December 2023, each Bitcoin was worth more than 17,000 US dollars.

Due to the success of the Bitcoin system, more and more people are investigating blockchain technology. Ethereum is another famous permissionless blockchain system that was proposed by Vitalik Buterin [18]. The value of an Ether, which is the digital coin in the Ethereum system, was about 160 US dollars on March 18, 2020. In comparison to the Bitcoin system, its innovation is the smart contract system [26] that works on top of the blockchain. The transaction in the Bitcoin system involves a stack-based script. But due to its simplicity, it can only perform limited operations. However, a smart contract is a Turing complete program that runs on Ethereum virtual machine (EVM). This gives the system more functionality instead of just being a cryptocurrency like Bitcoin. Also, Ethereum employs account based ledger in comparison to the transaction-based ledger in Bitcoin. Another feature of Ethereum is that it can also be deployed as a permissioned blockchain. Ethereum switched from PoW to PoS in 2022.

Algorand is an under-development permissionless blockchain that leading by a Turing award recipient, Dr. Silvio Micali, from MIT [35]. It is the first PoS-based blockchain

system. Algorand proposes a novel Byzantine agreement protocol to reach consensus on the transactions. This can make the system process 1000 transactions per second and confirm a transaction in seconds, which outperforms Bitcoin by a factor of more than 100.

2.5.2 Permissioned Blockchains

Hyperledger Fabric [5] is a permissioned blockchain system and one of the Hyperledger projects of the Linux Foundation. It is a modular open-source system that follows an execute-order-validate working flow which is different from others. All the other blockchains follow the order-execute architecture, which means transactions are being ordered first in a consensus setup and then executed sequentially in the same order in all peers. This will limit the throughput of the system due to the sequential execution. But execute-order architecture can execute transactions in parallel and as a result, increase the throughput. Another limitation of order-execute architecture is that some of the smart contracts need confidentiality of execution. As a result, Hyperledger Fabric proposed execute-order architecture. Moreover, Hyperledger Fabric also makes the consensus as a pluggable module. The lifecycle of the Fabric can be divided into three phases which are the execution phase, the order phase, and the validate phase. By using this infrastructure, the Hyperledger Fabric can achieve over 3560 [5] transactions per second. However, a major drawback of Fabric is that it currently uses Raft [81] consensus for ordering service, which is a crash fault-tolerant protocol. However, the development team of Hyperledger Fabric is working on a BFT-based ordering service for the next version.

Red Belly Blockchain [23] is a permissioned blockchain system supported by the University of Sydney and the Australian research council. It has the highest reported throughput, more than 600,000 transactions per second in one datacenter. Red Belly uses DBFT [21] to achieve consensus. DBFT is an efficient Byzantine consensus protocol using a weak coordinator to obtain excellent performance. Another feature of Red Belly is the sharded verification. Instead of verifying all the transactions in every participant, Red Belly only requires a subset of participants to perform the verification. This can optimize the usage of computational resources.

BoscoChain [100] is a developing permissioned blockchain based on the Bosco consensus protocol. Bosco is a one-step Byzantine fault-tolerant consensus protocol. It is easy to understand with a simple proof of correctness as well as high throughput and low latency. BoscoChain is also a sharded blockchain. A master ledger is in charge of not only maintaining the membership but also interleaving and integrating the participant ledgers.

2.6 Sharding

Besides the consensus of the system, sharding [41] is another intuitive approach to improve the scalability of the blockchain. Although sharding is a conventional technique to achieve high scalability in database systems, the effect of sharding in blockchain systems is not well understood. In a sharded blockchain, transactions are partitioned into multiple shards, and each shard builds a blockchain on its chunk of transactions. A sharded blockchain would ideally scale linearly with the number of shards in terms of throughput. However, if the shards are built with full sharding, which means a transaction is only recorded in one shard, as in Rapidchain [108], then cross-shard transactions need to be handled between shards. If not, as in BoscoChain [100], the shards must be interleaved to form a consistent final ledger. My research is focusing on sharding techniques in permissioned blockchains to increase system scalability.

In conventional blockchain systems, the server gathers transactions into a block and assembles the blocks as a single chain that is connected by the block hash. This entails reaching a consensus on the order of blocks. As explained before, the usage of a consensus protocol in this context limits scalability, and this observation has triggered a search for alternative designs that harness together multiple instances of a consensus protocol to increase transaction throughput. In particular, there is growing interest in sharding – a technique that distributes the storage, communication, and computation workload across multiple partitions (shards).

Sharding is a commonly used technique in traditional database systems where data are partitioned into multiple shards. Each shard is maintained on a separate server. By doing this, the load of the requests can be spread and balanced across all shards. Each shard represents a subset of data. For example, the data can be partitioned by the keys in the key-value store database. Each shard contains a subset of the key-value pairs. Different key-value pairs read and write can be operated concurrently in different shards without interference. However, the data in blockchain systems is a unique chain of blocks, where each block is interconnected with the hash of its previous block. Thus, how to partition the data remains the core problem of blockchain sharding. A major challenge is how to deal with cross-shard transactions, a transaction that the inputs and outputs locate in different shards.

Chapter 3

Performance and Fault Tolerance Trade-offs in Sharded Permissioned Blockchains

In this work, we illuminate some of the design trade-offs in sharded permissioned blockchains by examining the scalability of two methods for interleaving shards, which we refer to as strong temporal coupling (STC) and weak temporal coupling (WTC). STC and WTC in blockchains are introduced in our poster paper [67] at ICBC 2020. Nguyen's thesis [79] compares STC and WTC through experiments in a simulated environment and AWS EC2 in North American (NA) region. Beyond her thesis [79], since the naive STC cannot tolerate any failure, I implement and evaluate the performance of STC with a mechanism, called convergence module, for tolerating failures. The convergence module is first introduced in RCanopus [48] to tolerate network partitions. However, detailed algorithm design and evaluation are missing. In addition, I introduce a model to qualitatively analyze the performance of STC and WTC and conduct experiments in both single datacenter and globally distributed multiple datacenters. The results of the experiments match the prediction of the analysis.

3.1 Background and Related Work

In conventional blockchain systems, servers gather transactions into a block and assemble blocks into a single chain connected in a tamper-proof manner by the block hash. This

entails reaching consensus on the order of blocks, and thus deciding the predecessor of each block in the chain of hashes. The application of consensus in this context limits scalability, and this observation has triggered a search for alternative designs that harness together multiple instances of a consensus protocol to increase transaction throughput. In particular, there is growing interest in sharding – a technique that distributes the storage, communication, and computation workload across multiple partitions (shards).

Since blockchain systems process transactions in multiple stages, such as validation and ordering, sharding can be implemented in a variety of ways. OmniLedger [51], Elastico [61] and RapidChain [108] divide participants into committees, each comprising a subset of participants that work as a shard to process a subset of transactions. Committees are fixed for a period of time called an *epoch* and may be reconfigured at epoch boundaries. A special inter-shard commitment protocol is used to handle cross-shard transactions, which can be numerous due to hash-based transaction placement. OptChain proposes an algorithm that increases the performance of sharding via optimizing the placement of transactions into shards. BoscoChain [100] shards only transaction ordering and interleaves transactions explicitly into a master ledger to ensure that transactions are valid across different shards. Transactions can be validated partially by the shards, as in [23], but a global view of the master ledger is required to detect cross-shard inconsistencies such as double-spending. The master ledger is used to keep track of transactions to maintain the security of the system.

In general, sharding mitigates the scalability bottleneck of transaction ordering but complicates transaction validation. It can also weaken the resilience of a blockchain system to server failures by reducing redundancy, for example, the replication factor of transaction storage. Keeping the latter concern in mind, this work focus on the category of systems where designated participants merge transactions from all the shards to construct an explicit master ledger (e.g., [23, 100]).

The use of sharding to improve scalability in consensus protocols predates the rise of blockchains. Steward [3] is a Byzantine consensus protocol with a hierarchical structure, and uses a Paxos-like [55] leader-based consensus protocol to reach agreement across shards. BoscoChain [100] is a permissioned blockchain based on a similar structure and uses the leaderless Bosco [96] one-step Byzantine consensus protocol. Similarly to Steward, separate layers of consensus are used to order transactions first within each shard, and then globally. RCanopus [48] is a Byzantine hierarchical coordination protocol based on an earlier crash-tolerant system called Canopus [89]. Compared to Steward and BoscoChain, it dispenses with the top layer of consensus, opting instead for a simpler and more direct all-to-all broadcast. Blocks of transactions from different shards are interleaved implicitly in this approach, which requires that all shards produce blocks at roughly the same rate. If a

shard fails or is cut off by a network partition, liveness is restored by reconfiguring the set of shards carefully.

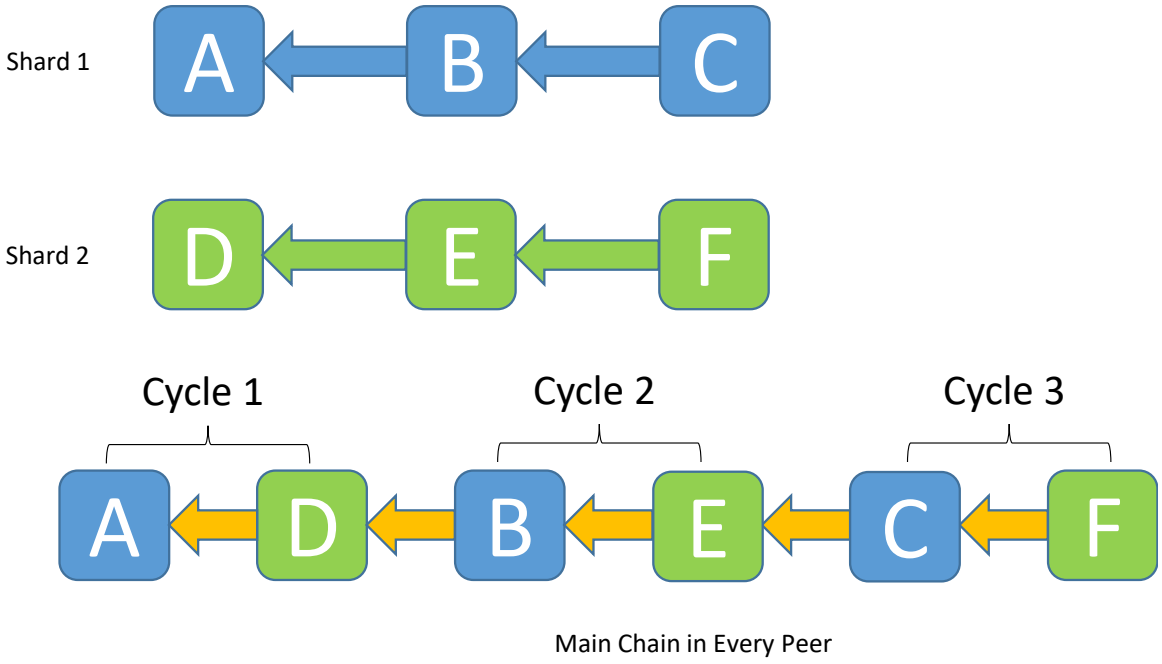
EPaxos [75] is a crash-tolerant consensus protocol that belongs to the Paxos family [55, 42], and can be used to order individual transactions or transaction blocks. In the crash failure model, consensus can be achieved with a majority of non-faulty nodes as long as the environment is sufficiently synchronous. In Paxos, commands must be handled by a leader in each cycle. If there are multiple leaders, the protocol may not make progress. Also, if the leader fails, the system needs to wait until a new leader is selected. Besides, the single leader limits scalability when there is a large volume of commands sent to the leader for processing. In contrast, EPaxos is leaderless, meaning that clients can send commands to any replica. This can relieve the high load on a single leader and thus achieve high throughput, especially in a geo-distributed environment. Due to its high performance and open-source implementation, we choose EPaxos for our prototypes to build shards as well as to interleave blocks. EPaxos source code in Golang is available to the public on GitHub [76].

3.2 Methods of Block Interleaving

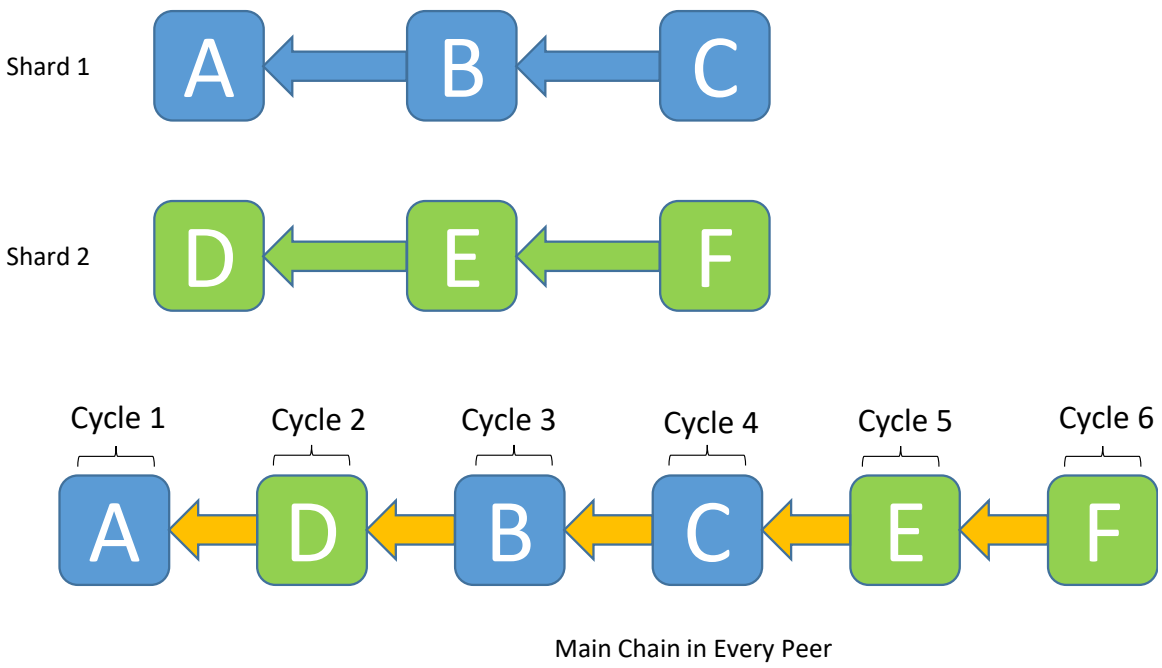
From a high-level point of view, *interleaving* in sharded blockchain systems means combining multiple sequences of blocks into a unique global sequence of blocks (i.e., main chain), and ensuring that each transaction remains valid in this global sequence. From this global view, validity is ensured straightforwardly by checking signatures and transaction inputs/outputs. The unique order of blocks must be computed either explicitly using an additional consensus protocol, or implicitly, for example by drawing blocks from different shards in round-robin order. An additional round of validation is then carried out, above and beyond the validation done by each shard, to counter malicious behaviours such as cross-shard double-spending. For example, in a cryptocurrency application, the system must prevent cross-shard double-spending (i.e., spending a coin more than once) and fake spending (i.e., spending a non-existent coin). This section explains two techniques used in recent research to implement the interleaving phase specifically. We refer to these as *strong temporal coupling (STC)* and *weak temporal coupling (WTC)* throughout the paper.

3.2.1 Strong Temporal Coupling

In the STC approach, blocks from different shards are interleaved based on a static ordering of the shards (e.g., in round-robin order), without explicit coordination. It is the simplest



(a) Strong Temporal Coupling (STC)



(b) Weak Temporal Coupling (WTC)

Figure 3.1: Two methods of interleaving shards in a blockchain.

interleaving strategy. Figure 3.1(a) gives an example of how two blockchain shards can be processed using STC. The order of blocks within each shard has already been decided, and now the two shards must be interleaved into the main blockchain. This is accomplished incrementally by processing blocks in *consensus cycles*. The advantage of this method is that each peer can do the interleaving independently without communicating with other peers, and still guarantee consistency if there is no Byzantine failure. The principal drawback of this method is that when one shard becomes unavailable (e.g., during a network partition), the entire system stalls. Multiple remedies to this problem are proposed in [48], all requiring that transaction commitment be delayed by one or more consensus cycles – a hazard to latency. Sacrificing fault tolerance for simplicity in the absence of failures, STC has the potential to outperform WTC in the common case, provided that transactions are distributed evenly across shards.

Since the simple design of STC cannot tolerate any shard failures, RCanopus [48] proposes a convergence module (CM) service to cope with network partitions. The CM acts similarly to an external group membership service that decides which shards participate in different consensus cycles, except that shards bypass the CM entirely in the absence of failures and compute the group membership implicitly by exchanging failure meta-data. A subtle mechanism is then used to synchronize implicit membership tracking in the failure-free case with explicit membership polling during network partitions and shard failures. This mechanism relies crucially on delaying transaction commitment by one consensus cycle, which increases latency but does not jeopardize throughput. Details are described in [48].

To enable a fair comparison with WTC, we consider two implementations of STC. The first implementation is naive in the sense that the system stalls when a shard fails. The second implementation is more elaborate and includes a portion of the Convergence Module service needed for our (failure-free) scalability experiments. We refer to these two implementations as STC-naive, and STC-CM, respectively.

In the naive implementation, an interleaving server requests ordered blocks from each shard, and then verifies the blocks sequentially according to the shard number, as shown in Figure 3.1(a). Algorithm 1 presents the pseudo-code for this implementation. In each cycle, each interleaving server waits for all shards to produce a new block (line 10). $S_j[cycle]$ is the block of shard j in a given cycle. Then, it appends the blocks in this cycle from every shard in the same order as line 11, which involves the procedure of appending the block to the main chain. The order must be fixed ahead of time and must be consistent across all the processes executing the procedure. Blocks are appended to the main chain in the same order, provided that they pass the final validity check (lines 2–4).

Algorithm 1: Strong Coupling

```
1  Procedure AppendBlock(block)
2      if verify(block) then
3          mainChain.append(block)
4      end
5  end
6  Procedure Interleave
7      cycle = 1
8      while True do
9          for each shard S_j do  // fixed order according to the shard number
10             wait until S_j[cycle] is known
11             AppendBlock(S_j[cycle])
12         end
13         cycle++
14     end
15 end
```

The more elaborate Convergence Module-based implementation is presented as Algorithm 2 and 3 and illustrates how shards exchange failure meta-data. First, the interleaving server generates a liveness report of each shard based on the cycle number at lines 19–26. The report records whether a shard appears alive or failed in that cycle. Next, the liveness reports are processed for each cycle at lines 30–51. In the failure-free case, the interleaving server receives reports from all shards indicating that an all-to-all broadcast was successfully completed in that cycle, and the interleaving server marks the corresponding blocks at lines 35–37 as the decision for that cycle. Otherwise, the interleaving server proceeds to lines 39–47, where it requests assistance from the CM for that cycle. The CM replies with the set S of shards that participated in that cycle. Finally, the corresponding blocks are interleaved at lines 56–58. The AppendBlock procedure is the same as Algorithm 1.

From Algorithm 1, we can see that there are two stages of processing in the naive implementation:

(1) extracting a new block from each shard, which can be considered transaction ordering, and

(2) validating and appending the block to the main chain, which can be considered transaction execution.

In comparison, Algorithm 2 and 3 introduce an additional stage to handle the liveness

Algorithm 2: Strong Coupling with CM

```
16 Procedure BroadcastReports
17     cycle = 1
18     while True do
19         for each shard S_j do
20             if S_j[cycle] received before timeout then
21                 report[cycle][S_j] = True
22             else
23                 report[cycle][S_j] = False
24             end
25         end
26         broadcast report[cycle] to all interleaving servers
27         cycle++
28     end
29 end
30 Procedure ProcessReports
31     cycle = 1
32     while True do
33         wait for report[cycle] from each interleaving server until a timeout
34         if received report[cycle] from every interleaving server and each report
            contains only True values then
35             for each shard S_j do
36                 state[cycle][S_j] = S_j[cycle]
37             end
38         else
39             request assistance from CM for cycle
40             retrieve missing blocks of cycle from shards set S decided by CM
41             for each shard S_j do
42                 if shard S_j is in S then
43                     state[cycle][S_j] = S_j[cycle]
44                 else
45                     state[cycle][S_j] = ∅
46                 end
47             end
48         end
49         cycle++
50     end
51 end
```

Algorithm 3: Strong Coupling with CM

```
52 Procedure InterLeave
53     cycle = 1
54     while True do
55         wait until state[cycle] has been computed by procedure ProcessReports
56         for each shard S_j do  // fixed order according to the shard number
57             AppendBlock(state[cycle][S_j])
58         end
59         cycle++
60     end
61 end
```

reports and, if needed, interact with the Convergence Module. According to the algorithms, the transaction throughput and latency can be qualitatively captured as:

$$l_{SC} = t_{rec\_max} + n_{shard} \times n_{txn} \times t_{exc} \tag{3.1}$$

$$l_{SC\_CM} = l_{SC} + t_{report} \tag{3.2}$$

$$tpt_{SC} = \min\{\frac{n_{shard} \times n_{txn}}{t_{rec\_max}}, \frac{n_{shard} \times n_{txn}}{n_{shard} \times n_{txn} \times t_{exc}}\} \tag{3.3}$$

$$tpt_{SC\_CM} = \min\{tpt_{SC}, \frac{n_{shard} \times n_{txn}}{t_{report}}\} \tag{3.4}$$

The notation used in the equations is explained in Table 3.1.

From equations 3.1 and 3.2, we see that the more elaborate CM-based implementation incurs a longer latency. Similarly, under the same allocation of resources, the CM-based implementation may exhibit lower peak throughput due to the additional cost of liveness report processing. However, the throughput of both implementations can also be limited equally by the transaction execution stage.

Notations	Description
l_{SC}	transaction confirmation latency of SC
l_{SC_CM}	transaction confirmation latency of SC_CM
tpt_{SC}	throughput of SC
tpt_{SC_CM}	throughput of SC_CM
t_{rec_max}	waiting time for a new block from the slowest shard in a cycle
n_{shard}	number of shards
n_{txn}	number of transactions in a block
t_{exc}	time to verify one transaction
t_{report}	time to broadcast and process the liveness reports

Table 3.1: strong coupling notations

3.2.2 Weak Temporal Coupling

In contrast to STC, WTC interleaves shards explicitly using another layer of consensus. Specifically, this is accomplished using a replicated state machine (RSM) that records a sequence of shard and block IDs, indicating the interleaving order. The procedure is illustrated in Figure 3.1(b). In the first consensus cycle, either block **A** or block **D** can be proposed, and the illustration shows that block **A** is decided. In the second consensus cycle, either block **B** or block **D** can be proposed, and block **D** is decided. The remaining blocks are ordered, validated, and appended accordingly, as shown in the illustration. WTC does not require all shards to grow uniformly or to participate in interleaving for that matter, but it does have to draw blocks from all available shards fairly to prevent starvation. Shard failure does not block the interleaving process.

Algorithm 4 is the pseudo-code of WTC. There is a dedicated thread running at each interleaving server that reads blocks from the shard it was assigned. The block is ordered through the consensus protocol at line 64, and the consensus protocol batches blocks internally for efficiency. Then, the block is stored along with its order in a data structure of pending blocks (line 65). The procedure INTERLEAVE executes transactions similarly to STC at lines 71–72. The transaction throughput and latency of WTC can be formalized as:

$$l_{WC} = t_{rec\_min} + t_{consensus} + n_{txn} \times t_{exc} \tag{3.5}$$

$$tpt_{WC} = \min\{\frac{n_{txn}}{t_{rec\_min}}, tpt_{consensus} \times n_{txn}, \frac{n_{txn}}{n_{txn} \times t_{exc}}\} \tag{3.6}$$

Algorithm 4: Weak Coupling

```
62 Procedure Ordering
63     for each block do
64         cycle = Consensus(block)
65         pendingblocks[cycle] = block
66     end
67 end
68 Procedure Interleave
69     cycle = 1
70     while True do
71         wait until pendingblocks[cycle] is set
72         AppendBlock(pendingblocks[cycle])
73         cycle++
74     end
75 end
```

The notation used in the equations is described in Table 3.2.

Compared to the two variations of STC, WTC is more sensitive to the performance of the consensus protocol because a single instance of this protocol is accessed at line 64, irrespective of the number of shards. This consensus protocol can potentially dominate the latency of WTC and limit throughput, making it inferior to STC. Aside from the consensus and transaction verification, the fundamental difference between STC and WTC is that the performance of WTC is determined by the aggregate shard growth rate, while STC is limited by the slowest shard. As the number of shards increases, the divergence in shard growth rate tends to increase as well, which could potentially give WTC an advantage over STC. Thus, although WTC requires an additional consensus stage, if a high-performance consensus protocol is used, WTC can potentially outperform STC even in the absence of failures, which is unexpected based on first impressions.

3.3 Prototype Implementation

Blockchain is a peer-to-peer ledger that is implemented following the established client-server model. In general, clients are processes that propose transactions to the servers, while servers are responsible for validating and storing the transactions in a chain of blocks.

Notation	Description
l_{WC}	transaction confirmation latency of weak coupling;
t_{rec_min}	time to extract a new block from any shard, which means the waiting time to read a new block from any shard;
$t_{consensus}$	time for ordering the block through the consensus protocol;
n_{txn}	number of transactions in a block;
t_{exc}	time for transaction verification, as in SC and SC_CM;
tpt_{WC}	throughput of weak coupling, which is the minimum of overall shard growing speed, the throughput of the consensus, and transaction validation speed.

Table 3.2: weak coupling notations

A server is a peer that carries out functions to process the transaction. Clients can communicate with any server without regard to the actual location of the data. Servers coordinate using a combination of state machine replication to record transactions in a fault-tolerant manner, and a broadcast protocol to disseminate blocks. Transactions are split into shards, and the shards are interleaved to compute a consistent ledger in all servers. In addition, in blockchain systems, digital signatures are used to validate transactions. This limits the power of a Byzantine peer to corrupt the system, and behaviors such as double-spending can be detected in the validating process.

A permissioned blockchain platform is implemented in Golang to compare the performance of STC-naive, STC-CM, and WTC. This section provides an overview of the prototype's system architecture, which is presented in Figure 3.2. Clients are at the top layer, which proposes transactions to the front end (FE) servers. Clients can only communicate with FE servers. The FE servers in the middle layer are used for coordinating between clients and shards, and also work as interleaving servers. These servers carry out three functions:

(1) collecting transactions from clients,

(2) verifying the signature and the existence of the received transactions, and

(3) gathering the transactions into blocks and distributing the blocks to shards.

Also, FE servers continuously extract blocks from each shard and interleave them into the final ledger through either strong coupling or weak coupling. The interleaving process involves the validation of double-spending against the interleaved blockchain. At last, each front end server stores a full copy of the interleaved final ledger, which is called a full node

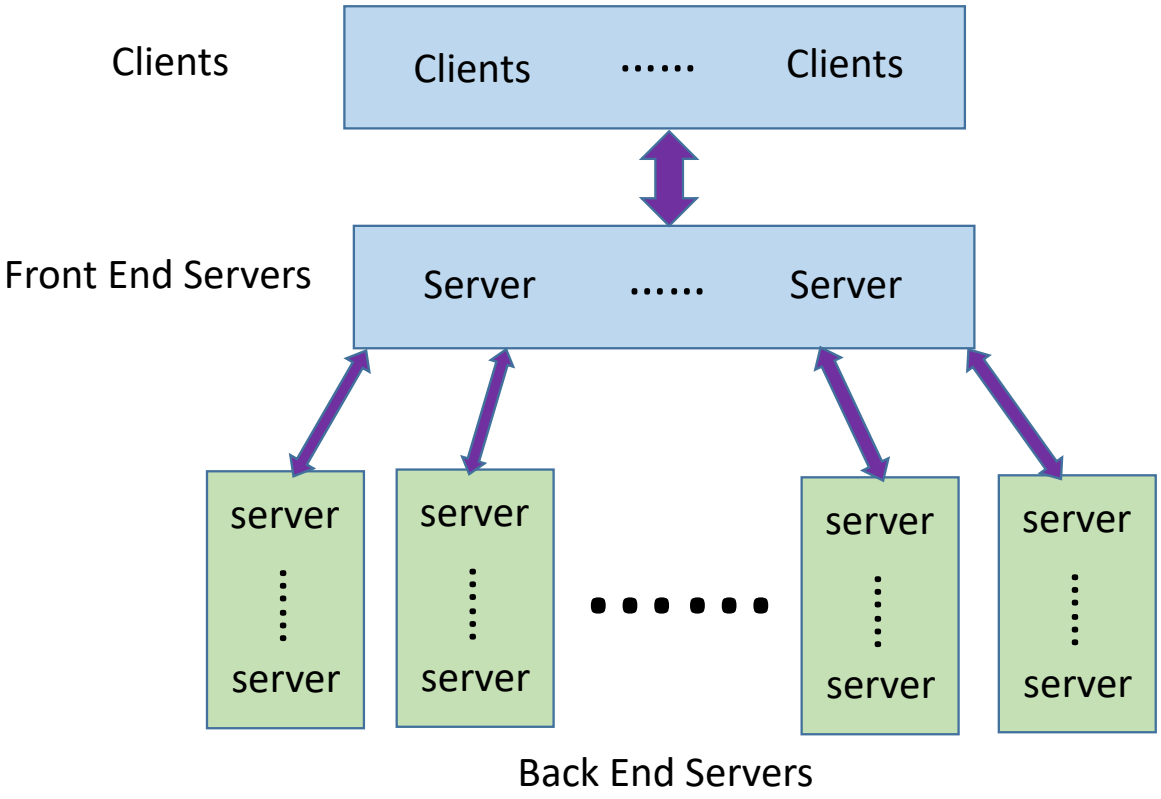


Figure 3.2: System architecture.

in Bitcoin. The bottom layer consists of back end (BE) servers, which are arranged into multiple shards. The main function of the BE servers is collecting blocks from FE servers and arranging them into a consistent chain of blocks as a shard, which involves consensus.

According to the architecture, the system is implemented with a high-performance crash-tolerant consensus protocol, EPaxos [75, 76]. Since the work is concentrated on the fundamental impacts of coupling methods, the high-performance consensus protocol lessens the impact of consensus on interleaving and shard building. Due to the popularity of the unspent transaction output (UTXO) model of Bitcoin, the workload of the system follows the UTXO model without scripts. Also, each transaction is merely a transfer of one or more unspent transactions from one use to another. A transaction is a data structure that includes a transaction id, the public key of the sender, a signature signed by the sender with its private key, an input, and an output. The input contains the hash of a previous unspent transaction and the sender address, which is the hash of the sender's public key. The output only contains the receiver's address. Moreover, each block includes

multiple transactions for better performance. And each block is connected by the hash of its previous block. We use the ECDSA algorithm as the signature scheme.

A sample workflow of a normal transaction is presented in Figure 3.3. The transaction starts with a user, Bob, who wants to send a coin to another user, Alice. Bob first proposes the transaction and then waits for the system to confirm the transaction. The FE server of the system is responsible for gathering the transaction and including it in the next proposed block. The FE server keeps proposing a new block to a shard for further processing. Meanwhile, the FE server extracts new blocks from the shards and interleaves all the new blocks into the main ledger with either strong coupling or weak coupling as introduced in the previous sections. After the FE server confirms that the transaction sent by Bob has been added to the main ledger, it informs Bob with a confirmation message. After this point, Bob can inform Alice regarding this transaction, and Alice can use this transaction as his coin in the future.

3.4 Evaluation and Discussion

The experiments are conducted in the Amazon EC2 environment to evaluate the relative performance of different block interleaving methods. Each process uses an m4.xlarge instance with four 2.4GHz Intel Xeon E5-2676 cores and 16GB of RAM, running Amazon Linux and Golang v1.13.6.

In each experiment, both the number of clients and the number of FE servers are equal to the number of shards, and each shard comprises 3 EPaxos replicas. Each FE server receives transactions from one client process and distributes blocks to one shard. Each client process runs multiple threads to mimic multiple users who propose transactions concurrently, where each thread represents an individual user of the system with a distinct key pair. We use closed-loop clients where each thread issues requests sequentially using synchronous calls. Throughput is controlled indirectly by varying the number of threads of each client. We measure the latency at the client on an end-to-end basis as the time from when a user submits a transaction to the time when that user receives a reply confirming whether or not the transaction is appended to the blockchain.

Prior to each experimental run, every user is assigned a wallet with some unspent coinbase transactions. These transactions are all valid coinbase transactions and are stored in their wallets. During workload execution, each user proposes a transaction by picking one coin from their wallet and sends it to a randomly chosen user simulated in the same client process. After the transaction is confirmed, the sender removes the input coin from its wallet and informs the receiver.

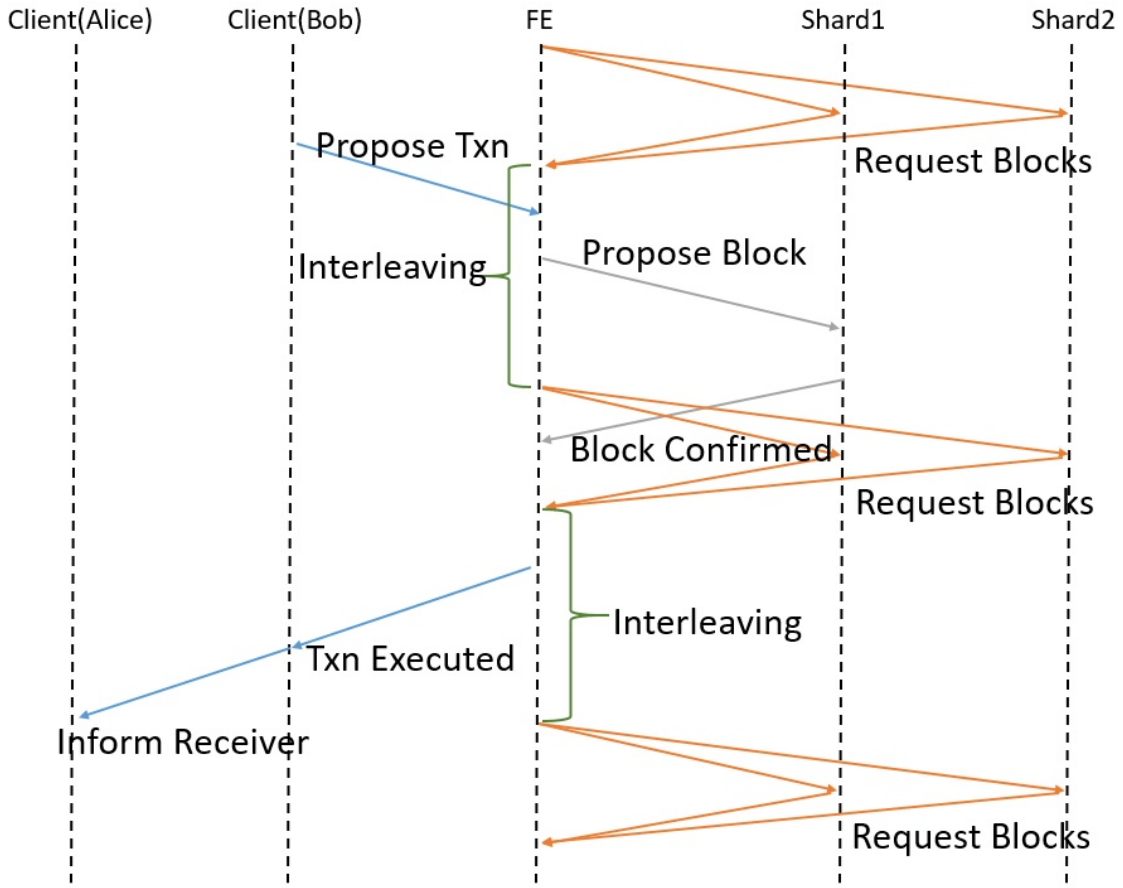


Figure 3.3: Transaction workflow.

In each experimental run, we allow the system to warm up for 5s, then start the measurement, and then keep the system running for another 5s after the measurement. For each run, we compute the median latency and the number of confirmed transactions per second (i.e., throughput) over a 20s measurement window. We repeat each experiment 3 times and plot the sample average as the data point in our graphs.

3.4.1 Single Datacenter Experiments

The entire system runs in the US-EAST (Ohio) region, where the network latency is less than 1 ms. This configuration minimizes the overhead of the consensus protocol and allows

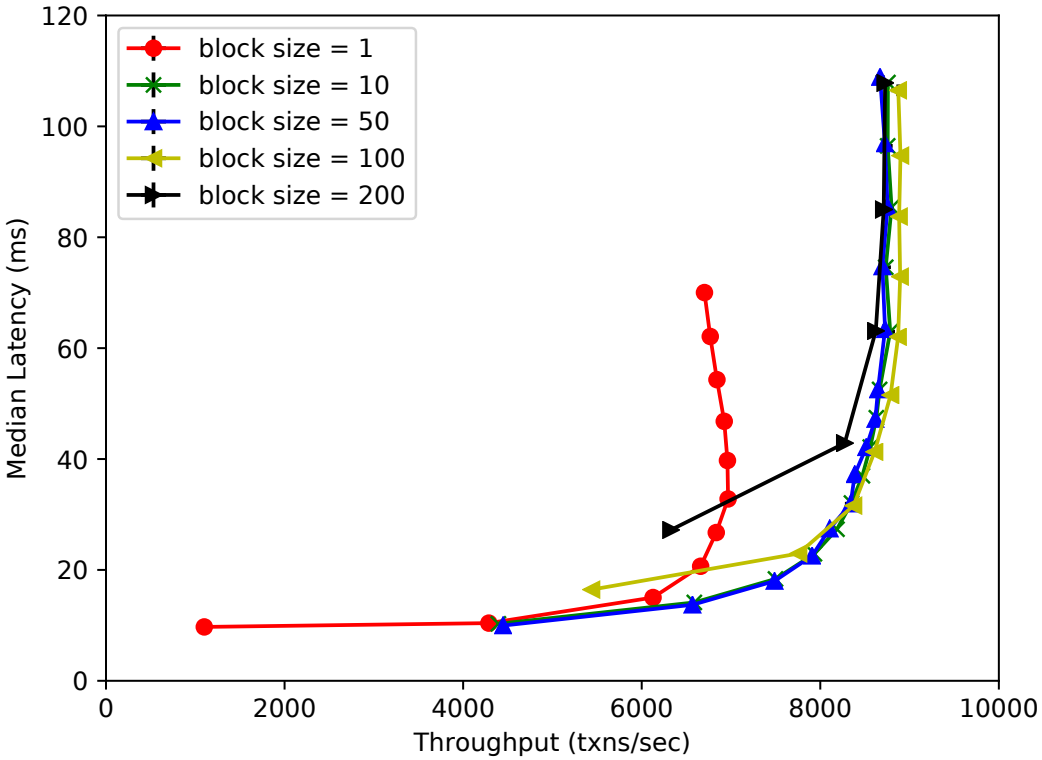


Figure 3.4: Performance of one shard with different block sizes.

us to examine other performance-limiting factors.

Prototype Fundamental Performance. In the first experiment, we establish a performance baseline by evaluating a single shard. This case yields a baseline where no interleaving is performed; WTC and STC collapse to a single protocol. Since the size of the block is a factor that affects latency and throughput, we investigate the impact of this parameter. In general, larger blocks tend to increase the transaction latency, while small blocks tend to limit the system throughput.

In the experiment, we vary the number of transactions in a block to determine a good block size. From Figure 3.4, we see that peak throughput varies between 6,500 txns/s and 9,000 txns/s. Throughput improves initially as the block size increases from 1 to 10-50. As the block size increases from 50 to 200, peak throughput remains steady but

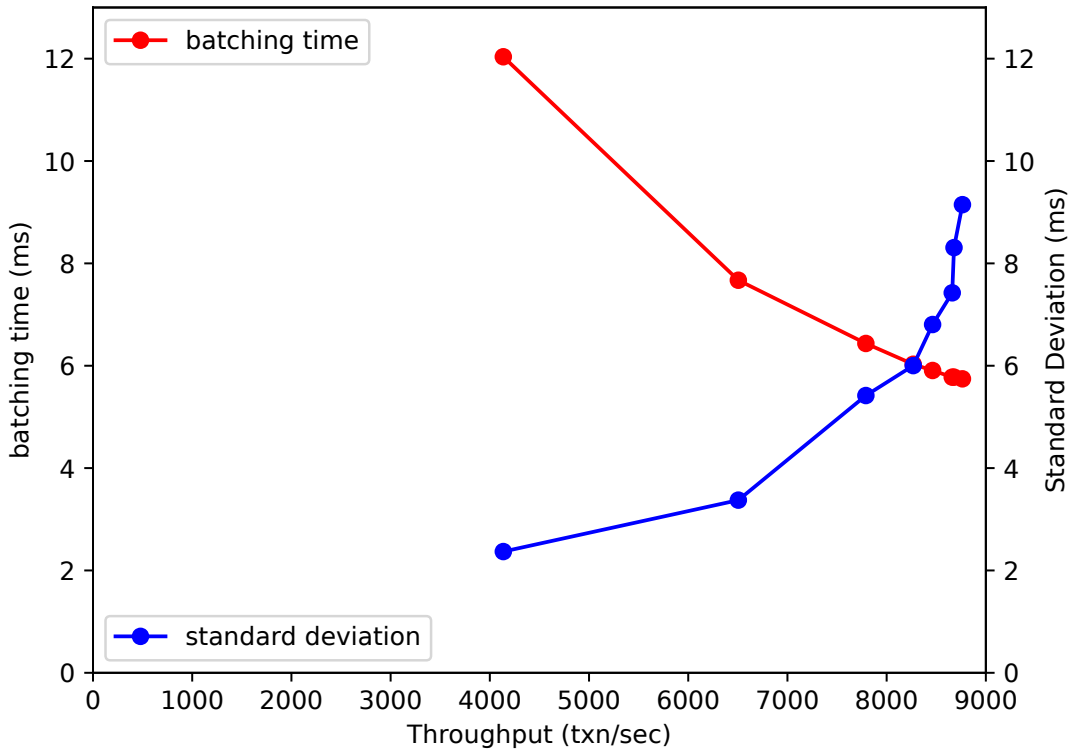


Figure 3.5: Batching time.

latency increases. Thus, a block size of 10-50 yields the best latency-throughput trade-off in this experiment. We adopt a block size of 50 for subsequent experiments and accordingly consider 9,000 txns/s an approximate upper limit on the throughput of one shard.

Our second experiment in one shard case measures the batching time, which we define as the time required to fill a block of constant size. This is an important metric since the batching time affects the shard growth rate in both STC and WTC. From Figure 3.5, we can see that the batching time (red) decreases as throughput increases, which is expected. However, the variation in the batching time (blue) increases sharply. This indicates that, even with constant block size, the time to obtain a new block exhibits substantial variation at high levels of throughput. This occurs due to competition for resources as we approach the system's performance envelope. In the worst case observed in this experiment, the standard deviation of the batching time approaches 10ms, which exceeds the single data-

	Throughput(txn/sec)	Latency(ms)	Notes
Estimated	193.42	5.17	Block size=1 1 client sends transactions continuously.
Measured	193.4	5.06	
Error	<1%	2%	
Estimated	4432.6	11.28	Block size=50 50 clients send transactions concurrently.
Measured	4432.5	9.94	
Error	<1%	12%	

Table 3.3: Accuracy

center network latency of around 5ms. Thus, under high throughput, the system inherently exhibits divergence of the shard growth rate.

To examine how accurate the model of both the throughput and latency is, I conduct an experiment in a 1-shard case to compare the measured values to the estimated values as Table 3.3. Since the model requires that each stage is running with dedicated resources, I measure the performance under low throughput where the resource is enough for each stage. Under high throughput conditions, each stage is competing CPU and network resources. The estimated throughput and latency are calculated by measuring the average time interval of each block and the average time to execute a transaction according to equations 3.1 and 3.3. I compare the results under two cases, one is that each block only includes 1 transaction, and only 1 client sequentially sends transactions. In this case, both the estimated throughput and latency are almost the same as the measured throughput and latency. When I increase the block size to 50 and require 50 clients to send transactions concurrently. The error of the throughput is still less than 1%. But the latency includes overhead coming from the clients (preparing the transactions) and the servers (collecting transactions into a block). In summary, the model can accurately estimate the throughput while the latency may suffer overheads.

Scalability. In this experiment, I investigate scalability by measuring the throughput and latency of three and five shards and comparing them against the earlier one-shard experiment. From Figure 3.6, STC-naive and WTC achieve similar peak throughput and latency with three shards, while STC-CM has slightly higher latency and lower peak throughput. Error bars indicate the sample standard deviation, which is only visible at high throughput. This observation reflects the fact that STC-CM requires additional time to receive and process liveness reports, and also indicates that the additional layer of consensus in WTC is not a performance limiting factor in a single datacenter. However, when the number of shards increases to five, we see a small separation between STC-naive and WTC, the latter taking the lead. This corroborates our earlier prediction that WTC can

outperform STC when shards grow unevenly – an effect whose intensity is amplified as we increase the number of shards.

Since we observe substantial gains in throughput with the addition of shards, the scalability experiment also proves that the system is not bottlenecked by the final transaction validation phase, where blocks are checked sequentially after being ordered by the shards and then interleaved by the front-end servers.

Our overall conclusion from the single datacenter experiments is that the interleaving approach to sharding a blockchain is scalable, though sub-linearly. The peak throughput increases from roughly 9,000 txns/s at one shard, to more than 20,000 txns/s at three shards, and more than 30,000 txns/s at five shards. In a system with three shards and no interleaving, the ideal peak performance should be $9{,}000 \times 3 = 27{,}000$ txns/s. Similarly, at five shards we expect roughly $9{,}000 \times 5 = 45{,}000$ txns/s. Thus, the overhead of interleaving in terms of throughput is approximately 25% at three shards and 30% at five shards. Additionally, we observe that STC-CM suffers a substantial latency overhead. As a result, the interleaving approach – strongly or weakly coupled – impacts scalability noticeably. Based on the algorithm analysis, the overhead comes from the additional consensus in WTC, as well as the final transaction validation performed in the interleaving servers.

3.4.2 Multi-Datacenter Experiments

In this suite of experiments, the system is deployed across multiple AWS EC2 datacenters located on different continents: Ohio (OH), São Paulo (SP), Frankfurt (FR), Oregon (OR), and Tokyo (TO). The network latency among these datacenters is quantified in Table 3.4, averaging approximately 150ms. We run the experiments using three and five shards to evaluate how the high latency of the wide-area network affects the performance of different interleaving methods. Each datacenter hosts one shard and one FE server, as well as one client process. The client only proposes transactions to the FE server in the same datacenter. Three-shard experiments use the OH, SP, and FR datacenters, whereas all five regions are used for five shards.

Figure 3.7 presents the scalability of the system with three and five shards. We observe a clearer separation in terms of throughput and latency between STC-naive, STC-CM, and WTC. STC-naive achieves the highest peak throughput and lowest latency. Although its latency increases several-fold as compared to the single datacenter experiments (i.e., 100-150ms vs. 15-20ms at low-to-mid throughput), peak throughput declines only by roughly 10% at five shards. This is because the performance of STC-naive and STC-CM is limited by the CPU instead of the network. In addition, STC-CM suffers from the

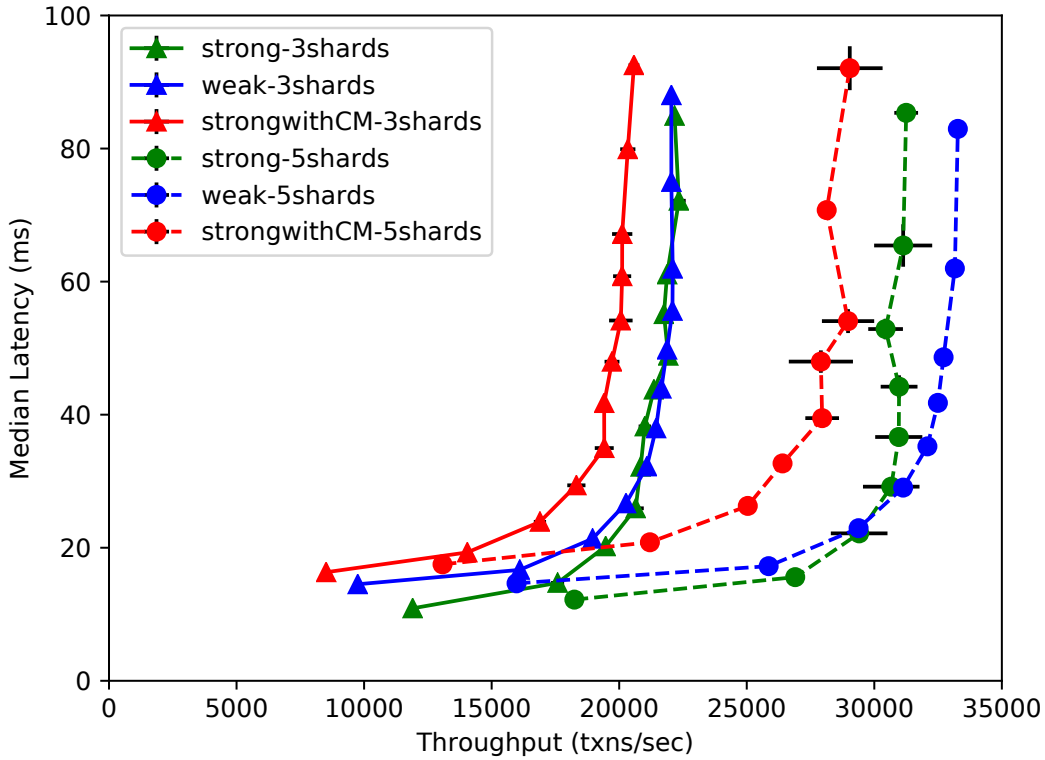


Figure 3.6: Scalability using a single datacenter.

overhead of broadcasting and processing liveness reports, which adds 100-200ms of latency, or the equivalent of one network single trip. Thus, STC-CM performs noticeably worse than STC-naive, but is still much better than WTC, which not only exhibits much higher latency but also lower peak throughput. WTC incurs the largest penalty in the multi-datacenter scenario as compared to a single datacenter because the additional layer of consensus used to interleave blocks from different shards operates across datacenters, and is affected heavily by the wide-area network latency. This is in contrast to the consensus used to record blocks in each shard, which is deployed over multiple servers in the same datacenter. As a result, consensus becomes a bottleneck in WTC, which lags noticeably behind STC despite uneven shard growth.

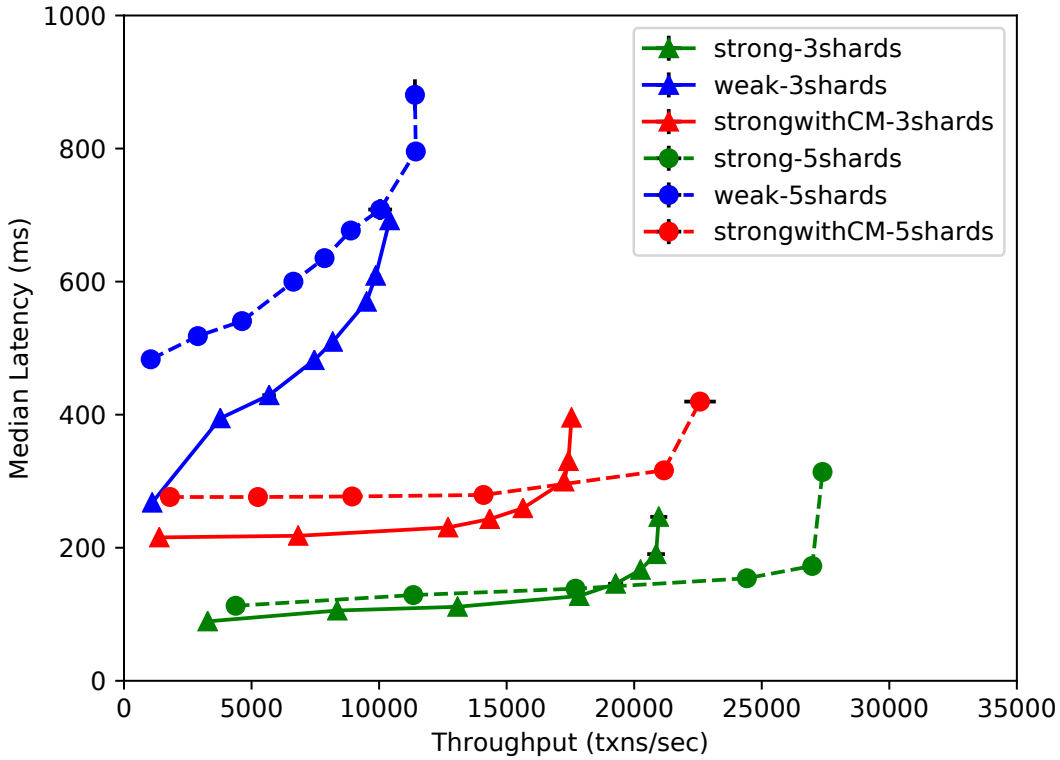


Figure 3.7: Scalability using multiple datacenters.

3.4.3 Discussion

The experiments demonstrate that when operating over a low-latency network, WTC can outperform STC-naive despite doing additional work to interleave blocks from different shards. This is a somewhat surprising outcome given the large discrepancy between these two designs in terms of tolerating network partitions and shard failures, where WTC is far more robust than STC-naive. One way to interpret this result is that WTC is robust not only to failures but also to stragglers that arise from uneven shard growth even during failure-free operation. We also compared WTC against the more elaborate STC-CM implementation, which tolerates network partitions at the cost of delaying transaction commitment. Such a comparison levels the playing field between strong and weak coupling and captures more precisely the cost of interleaving transactions from different shards. The

Latency (ms)	OH	SP	FR	OR	TO
OH	-				
SP	123	-			
FR	99	200	-		
OR	48	170	140	-	
TO	156	251	267	101	-

Table 3.4: Round-trip network latency between datacenters.

single datacenter experiments show similar throughput between STC-naive and STC-CM, where STC-naive achieves lower latency, and neither variation of STC beats WTC for throughput. However, in a multi-datacenter deployment, both variations of STC outperform WTC, which is hampered by the overhead of consensus across a wide area network.

Regarding fault tolerance, STC-naive cannot tolerate the loss of even one shard, for example, due to correlated crash failures or a network partition. If any shard becomes unavailable, STC-naive stalls entirely until this shard become responsive once again. As a result, the STC-naive is not a practical approach to be used independently. However, STC can be a good strategy if used in tandem with other techniques, like the Convergence Module in RCanopus [48]. In contrast, WTC deals with both failures and stragglers naturally by using consensus to decide which shard contributes the block for a given position in the main chain.

3.5 Conclusion

In this work, we compared two methods for interleaving transactions from different data partitions, or shards, in permissioned blockchains. We refer to these as strong temporal coupling and weak temporal coupling, reflecting the restriction they impose on the rate of progress at different shards. We clarified these methods and presented an analysis of each approach. Our experimental results prove that weak coupling achieves better scalability than strong coupling in a single datacenter, while strong coupling performs much better in a geo-distributed multi-datacenter environment, in accordance with our earlier analysis and prior work [79]. On the other hand, the strong coupling is inherently less robust against network partitions and shard failure, where the weak coupling is able to maintain progress naturally. This drawback can be mitigated by introducing an optimization, the CM [48], that sacrifices some of the peak throughput and latency while maintaining a lead

over weak coupling in the geo-distributed case.

Due to the superiority of STC with CM in shard interleaving, we elaborate on it in solving the consensus problem in the next chapter. We first formalize the consensus problem and then use the idea of STC with CM to develop the algorithms to solve the problem under the crash failure model and Byzantine failure model, respectively.

Chapter 4

Antipaxos: Taking Interactive Consistency to the Next Level

Although there exist many consensus protocols, most of them are primarily concerned with handling failures and competing proposals instead of scalability. This is accomplished through complex message passing and processing, which imposes overhead. Notably, the problem is traditionally solved using single-leader quorum-based techniques, such as variations of Lamport's Paxos protocol [55]. In such protocols, the leader plays a central role and limits scalability. Using these protocols to implement a replicated state machine (RSM) is inherently costly as every batch of state transitions has to be proposed by the leader and requires at least one dedicated consensus cycle.

We first define *k-IC*, a multi-value consensus problem. And then we propose Antipaxos (AP), a novel mechanism for reaching agreement efficiently on a collection of proposals to solve *k-IC* problem under failure-free execution, that resorts to classical consensus only when a failure occurs. The key idea is to broadcast a liveness report to everyone to decide whether a failure has occurred. If a failure occurs, an assisting system, Decision Module (DM), is introduced to handle the failure cases. AP elaborates on the design of STC with CM in the previous chapter in making consensus. We extend the CM as DM to tolerate crash failure and Byzantine failure in making consensus, respectively. The original design [48] requires the CM nodes to actively request reports from the RCanopus system. AP eliminates this without compromising the safety and liveness properties. In addition, the original design of CM requires delaying the report multiple cycles which incurs latency overhead. We decouple the reports by using an independent message to reduce latency. The drawback is to handle more smaller messages. We also prove the safety and liveness properties. In the common case, Antipaxos is able to commit n batches of proposals in each

consensus cycle using $\Theta(n^2)$ messages, where n is the number of replicas. Furthermore, there is no single designated leader that synchronizes the protocol. In addition, consensus cycles can be pipelined for higher throughput in a wide-area network (WAN).

4.1 Model

To achieve *k-IC*, we need a message-passing system for a set of processes to reach agreement on a list of values. We assume a reliable point-to-point network that connects any pair of processes through separate channels to send and receive messages. The channel cannot lose, corrupt, duplicate, or reorder messages. Regarding synchrony, the practical environment may behave differently in different time periods. To best fit the environment, we assume that the environment has two modes: synchronous mode and asynchronous mode. Normally, all the processes and the network operate in the synchronous mode, in which there is a known upper bound on the message and processing delay. If the delay exceeds the upper bound either due to asynchrony or process failure, the environment turns to the asynchronous mode. Since the consensus problem may not be solvable in a fully asynchronous environment in case of a failure, we assume the existence of a classical consensus protocol that guarantees safety while the liveness is maintained under sufficient synchrony assumption. This protocol is used to implement a crucial building block of Antipaxos, which not only handles failure cases but also ensures a consistent view of the protocol across synchronous mode and asynchronous mode. Considering fault tolerance, our model assumes a bounded number of faulty processes, which may experience either crash failures or Byzantine failures, as explained in Section 4.3 and Section 4.4, respectively.

4.2 k-Interactive Consistency

Classical consensus reach agreement on a single proposal at a time, which limits scalability. If an RSM is implemented with this kind of consensus protocol, every state transition would require at least one distinct consensus cycle to reach agreement. Each consensus cycle is a complete round of consensus. To achieve high scalability in a wide-area network (WAN), we introduce "k-interactive consistency" which aims to reach agreement on an ordered list of proposals from different processes within a single consensus cycle. A solution to this problem can be used to implement a scalable RSM in the WAN. *k-IC* is a combination of Interactive Consistency (IC) [83] and Vector Consensus (VC) [29], which has a parameterized number of proposals in different situations. IC was first proposed to tackle certain

problems in synchronized fault-tolerant systems where failure can be detected easily, such as synchronization of clocks and stabilization of input from sensors. Due to the synchronized environment, IC guarantees to reach agreement on at least $n - f$ values, where n is the total number of processes and f is the number of faulty processes. VC proposes to reach agreement on a vector of values in an asynchronous setting and assures there are at least $f + 1$ *non-null* values in the vector that corresponds to non-faulty processes. To fit different synchronization situations in one environment, we propose *k-IC*. Optimally-resilient IC protocols [11] solve the IC problem by running multiple instances of consensus protocols and selecting one as the final decision. Asynchronous IC is achieved in [63].

We now present the definition of *k-IC*. To simplify the description, we assume that each proposal is a value. Instead of agreeing on a single value, we modify the classical consensus problem so that a set of n processes, $P = \{P_1, ..., P_n\}$, agrees on an ordered list of n values, V, which tries to include the proposal value v_i of each process P_i but may also return *null*. The order of these values is predetermined, such as ordered numerically by the process ID, $V = [v_1, ..., v_n]$. A *null* value of process P_i indicates that P_i may have failed. Crash failure and Byzantine failure are two failure models when designing a consensus protocol [101]. Crash failure means that a process fails by halting, but it is working correctly before the halt. Byzantine failure means that a process can crash, delay sending messages, or send arbitrary messages, which means the Byzantine process can have arbitrary behaviours. A solution to the *k-IC* problem in the crash failure model must satisfy the following properties:

Validity (Safety 1): If a process decides a *non-null* value v_i in V, then v_i was proposed by process P_i.

Agreement (Safety 2): No pair of processes ever decides different V.

k-Completeness (Safety 3): The number of *null* values in V is no greater than the non-negative integer k.

Termination (Liveness): Every non-faulty process eventually decides a V.

The problem specification under Byzantine failures shares the same k-Completeness and Termination properties as above, but has different validity and agreement:

Validity (Safety 1): If a non-faulty process decides a *non-null* value v_i in V and process P_i is not faulty, then v_i was proposed by process P_i.

Agreement (Safety 2): No pair of non-faulty processes ever decides different V.

The k-Completeness is a parameterized property which rules out the trivial case where the decision values are all *null*. Without this property, a list of n *null* values would also

satisfy the requirements of k-*IC*. Note that a system may satisfy k-Completeness with different k in different situations, including $k = 0$ under favourable conditions.

By rotating the proposers (e.g., in round-robin order of the processes), if the proposer does not fail and proceeds in time, the classical consensus protocol can reach k-*IC* in multiple rounds of consensus by agreeing on one value per round. Otherwise, the solution becomes complex, and additional mechanisms have to be carried out to handle the failure or asynchrony of the proposers.

4.3 Crash Fault-Tolerant Design

In this section, we introduce the design of AP under the crash failure model. A crash failure means that a process halts and no longer processes or replies to messages. We use message and process delay as unreliable failure detectors. This means that if a timeout occurs either due to a process crash or asynchrony, one process suspects that the other process has failed.

Suppose there are $2f + 1$ processes that would like to reach k-*IC*, where at most f processes may crash at any time. We assume that if a process crashes, it will not be able to recover until the end of the consensus cycle. If there is no timeout, the consensus must include one *non-null* value from every process. Note that *empty* is a special *non-null* value, indicating that no value was received from the client process. The order of these values is predetermined in a round-robin order of the processes. The goal is to solve k-*IC* with a fast path solution in the absence of failure and asynchrony, falling back on a slow path to handle other cases. In addition, it is important to ensure consistency between the fast path and the slow path.

4.3.1 Overview

The design involves three stages: propose value, broadcast report, and commit. Figure 4.1 illustrates the message flow of the crash failure design. The first two stages require each process to send a message to all other processes. In the absence of a timeout, the commit phase does not require any extra communication, and each process proceeds to commit and execute the decision by itself. As a result, AP realizes consensus on a collection of n values using $\Theta(n^2)$ messages in one network round trip. If a failure or asynchrony occurs, leading to a timeout, the affected processes seek assistance from the Decision Module (DM), which is a fall-back RSM implemented with a conventional consensus protocol, as the slow path

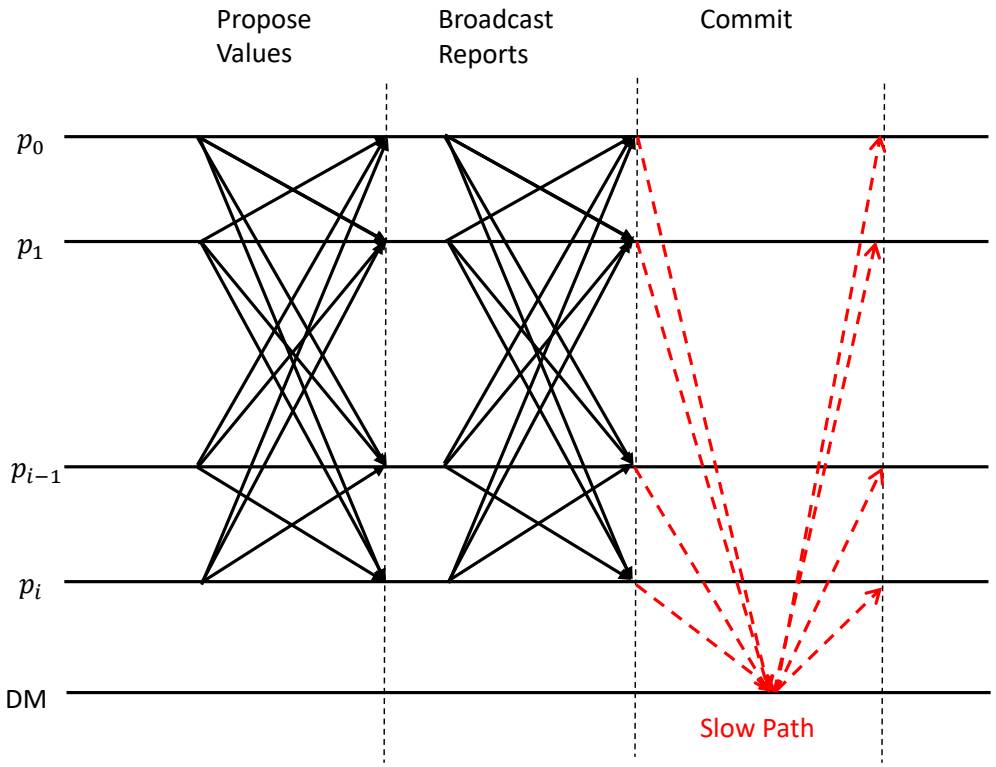


Figure 4.1: Example of AP Message Flow.

in the commit phase. More importantly, some processes may proceed on the fast path, but others proceed on the slow path. Thanks to the design of the report, the DM determines the same result as the processes in the fast path, thus the algorithm ensures a consistent view under this situation as well.

4.3.2 Algorithm

Algorithm 5 is the main procedure running in each process. As mentioned earlier, the protocol is divided into three stages. In the propose value stage, each process P_j sends its proposed value v_j to all other processes. Each process must wait for at least $f + 1$ values (including its own) before proceeding to the next stage, since that many processes

are guaranteed to be non-faulty, and waits for f additional values until a timeout occurs. Since there are at most f out of $2f+1$ processes that may fail, it can be guaranteed that at least $f+1$ processes successfully send the proposed value. Then, in the broadcast report stage, each process P_j prepares and sends a liveness report rep_j to all processes to indicate the status of other processes according to whether the value has been successfully received at lines 82 to 88. If process P_j receives a *non-null* value v_i from process P_i, the report records the value $rep_j.v[i]$ of P_i as v_i. Otherwise, the report marks the value of P_i as *null* which means process P_j did not receive a value from process P_i before the timeout and so P_j suspects that P_i has failed. Before continuing to process the reports, each process waits until it receives at least $f+1$ *rep* before the commit stage (includes itself), and f more until a timeout occurs. From line 94 to 104, the process derives V, which is the outcome of *k-IC*, according to the received reports. In the common case, each process receives $2f+1$ reports without any *null* value, which indicates that all processes are alive and working normally. As a result, the result contains the values proposed by all processes. At this point, *k-IC* is achieved, and every process agrees on the same list of values.

However, if any timeout occurs, we use another mechanism called the Decision Module (DM) to reach the *k-IC*. The DM is a set of DM nodes, where each node is co-located with an RSM replica for fault-tolerant distributed coordination. State changes are achieved by submitting requests from the DM nodes to its co-located RSM replica. The RSM makes at most one state change in one consensus cycle. By using this design, we ensure that any two processes determine the same list of values, which we prove in section 4.3.4.

Algorithm 6 describes the procedure running in each DM node. If an AP process P_j receives any *rep* from any process that reports a failure, then P_j sends its report rep_j to the DM and request assistance. Upon receiving enough reports from the AP processes, the DM produces a final report $final\_rep$ according to the received reports. The final report is derived as follows: if all the reports until a timeout state that process P_i has failed, then the final report uses *null* as the value of the process P_i. Otherwise, the value of process P_i is marked with its true value v_i. Next, the DM sends the $final\_rep$ to RSM to record the decision. The *Execute* function in Algorithm 6 is the state transition procedure of the RSM. If the RSM is constructed using a single leader consensus protocol, the leader responds to handling the $final\_rep$. Otherwise, every DM node must produce and suggest a $final\_rep$ to tolerate DM node failure. For each consensus cycle, the RSM has to only record the first $final\_rep$ as the decision and ignore the others, where Dec is the decision in the RSM. This ensures that all processes receive the same decision from the DM. Finally, the DM replies the decision to the AP processes.

To summarize, Antipaxos uses communication success to determine the set of processes that are considered alive. It broadcasts this information to all processes, and then each

Algorithm 5: Design under crash failure

```
76  Procedure AP_Propose P_j(v_j)
77  |  send v_j to all the processes
78  end
79  Procedure AP_Report P_j(v_j)
80  |  wait for at least f + 1 proposed values
81  |  wait for f more values until a timeout
82  |  for each process P_i do
83  |  |  if v_i received then
84  |  |  |  rep_j.v[i] = v_i
85  |  |  else
86  |  |  |  rep_j.v[i] = null
87  |  |  end
88  |  end
89  |  send rep_j to all the processes
90  end
91  Procedure AP_Commit P_j(v_j)
92  |  wait for f + 1 rep
93  |  wait for f more rep until a timeout
94  |  if 2f + 1 reports were received and they all indicate the same non-null value for
    |    each process then
95  |  |  for each process P_i do
96  |  |  |  V[i] = v_i
97  |  |  end
98  |  else
99  |  |  send rep_j to DM and request assistance
100 |  |  wait for the decision Dec from DM
101 |  |  for each process P_i do
102 |  |  |  V[i] = Dec.v[i]
103 |  |  end
104 |  end
105 end
```

Algorithm 6: DM main loop under crash failure

```
106 Procedure DM P_m
107     wait until the first rep received
108     wait for more rep until a timeout occurs
109     if received some rep with null value then
110         wait until at least f + 1 reports
111         wait for f more rep received until a timeout
112     end
113     for each process P_i do
114         if at least one report rep_x state rep_x.v[i] = v_i then
115             final_rep.v[i] = v_i
116         else
117             final_rep.v[i] = null
118         end
119     end
120     send final_rep to RSM
121     read decision Dec from RSM
122     send Dec back to the processes who requests assistance
123 end
124 Procedure Execute P_m(final_rep)
125     if Dec is not decided yet then
126         Dec = final_rep
127     end
128 end
```

process excludes the proposed values from any processes that the DM has identified as having failed. In the common case, k-IC is achieved with $2n^2$ messages, which amounts to $2n$ messages per value. This is less than Paxos [55] and even the fast path of EPaxos [75]. Regarding the latency, AP in the case of the fast path only requires one round-trip of network delay, while the other cases require two round-trips plus the processing time of the DM.

4.3.3 Running Examples

As a running example, consider the case of three processes: $P1$, $P2$, and $P3$, which propose values $v1$, $v2$, and $v3$, respectively. In this example, $f = 1$ and there are $2f + 1 = 3$ processes.

Case 1: Suppose that $P1$ receives inputs $v2$ and $v3$ from $P2$ and $P3$, respectively. Similarly, both $P2$ and $P3$ receive inputs from other processes. Then the following reports are produced by each process:

$P1 : \{P1 : v1,\ P2 : v2,\ P3 : v3\}$

$P2 : \{P1 : v1,\ P2 : v2,\ P3 : v3\}$

$P3 : \{P1 : v1,\ P2 : v2,\ P3 : v3\}$

If all the processes receive these reports, they determine the list of values $\{v1,\ v2,\ v3\}$ without seeking assistance from the DM as line 95 to 97.

Case 2: If $P2$ fails with respect to both $P1$ and $P3$, the reports become:

$P1 : \{P1 : v1,\ P2 : null,\ P3 : v3\}$

$P2 : Failed$

$P3 : \{P1 : v1,\ P2 : null,\ P3 : v3\}$

$P1$ and $P3$ seek assistance from DM by sending its report. As DM detects that $P2$ has failed, the non-faulty set of processes becomes $P = \{P1 : v1,\ P3 : v3\}$.

Case 3: However, if $P2$ fails with respect to $P3$ only, the reports become:

$P1 : \{P1 : v1,\ P2 : v2,\ P3 : v3\}$

$P2 : Failed$

$P3 : \{P1 : v1,\ P2 : null,\ P3 : v3\}$

Then, upon receiving the reports, both $P1$ and $P3$ send their report to the DM and wait for the decision as line 99 to 103. Since there is one report indicating that $P2$

has value $v2$, the DM still considers $P2$ as alive and replies to the set of processes as $P = \{P1 : v1,\ P2 : v2.\ P3 : v3\}$. Then both $P1$ and $P3$ decides the list as $\{v1,\ v2,\ v3\}$ directly.

4.3.4 Safety and Liveness

This section proves the safety and liveness of the protocol.

Lemma 1. *For each consensus cycle, there is exactly one decision made by the DM if at least one AP process seeks assistance.*

Proof. If at least one AP process submits an assistance request, the DM processes the report according to Algorithm 6 and sends the $final\_rep$ to the RSM. If the RSM has already recorded a decision, no more decisions are made. If a decision has not been made, then it makes a decision based on this request. □

Theorem 1. *Validity: If a process decides a* non-null *value v_i in V, then v_i must be proposed by the process P_i.*

Proof. There are two cases that a process determines the list of values V.

Case 1: No timeout occurred. All reports state the same *non-null* value v_i for process P_i.

According to the Algorithm 5 line 94, the value v_i must be sent by process P_i.

Case 2: Decision V is made by DM.

According to Lemma 1, there is only one decision V. If v_i is being included in V, it must be included in the report submitted by process P_x of AP. If P_x includes v_i in its report, it must receive v_i from process P_i.

As a result, in both cases, validity follows. □

Theorem 2. *Agreement: If any two processes determine a list of values, then both determine the same list of values.*

Proof. If any process determines a list of values, the outcome belongs to exactly one of the following two cases:

Case 1: The process receives $2f + 1$ reports, and none of them contain a *null* value. The outcome is the list without *null* value.

Case 2: The process either receives a report indicating certain *null* values of some processes, or it has not received $2f+1$ reports before the timeout. Then, the process sends a request to the DM and determines the outcome according to the decision made by the DM.

Therefore, if the outcomes of any two processes belong to Case 1, then each outcome is the list without *null* values, which is the same. If the outcomes of any two processes belong to Case 2, due to Lemma 1, the outcomes of these two processes are the same as well. Last but not least situation is that one outcome belongs to Case 1 and the other outcome belongs to Case 2.

Suppose that process $P1$ determines the outcome as Case 1, which is the list without *null* value, and another process $P2$ determines the outcome as Case 2. Then, the outcome of Case 2 is also the list without *null* value. To prove this, suppose for contradiction that the outcome of Case 2 includes at least one *null* value.

Since $P1$ determines the outcome as Case 1, it must receive $2f+1$ reports, and none of them have a *null* value. Now, consider how the DM makes a decision with *null* values. If the DM decides a process has failed, it must receive at least $f+1$ reports so that each report contains a *null* value of the failed process. However, $P1$ did receive $2f+1$ reports, which indicated that there are no malfunctions. Since there are at most $2f+1$ reports, this is a contradiction. As a result, if $P1$ determines the outcome as Case 1, the DM must determine a list without *null* values. Any process seeking DM assistance must reach the same outcome. Any process that does not need assistance from the DM determines the list without *null* value as well. Then the safety is proven. □

Theorem 3. *k-Completeness: The number of* null *values in* V *is not greater than* 0 *in the absence of failure and asynchrony. Otherwise, the number of* null *values is not greater than* f*.*

Proof. Similarly to Validity, there are two cases where a non-faulty process determines the list of values V.

Case 1: There is no message timeout. In this case, all reports state the same *non-null* value v_i for process P_i. As a result, there is no *null* value in V, which is 0-Completeness.

Case 2: Decision V is made by the DM.

The decision of the DM depends on the report submitted by each process. If any report contains a *non-null* v_i of P_i, then the decision value of P_i must be v_i as Algorithm 6 line 113 to 119. Since each report includes at least $f+1$ *non-null* values out of total $2f+1$ values, the total number of *null* values cannot exceed f, which is f-Completeness. □

Lemma 2. *If any process produces a report with at least one* null *value and seeks assistance from the DM, then at least f other processes also submit their reports and assistance requests to the DM.*

Proof. There are $2f + 1$ processes in total. Suppose that process $P1$ produces a report indicating that $P2$ has failed. Then, $P2$ either crashes when proposing a value and fails to submit a report, or it is merely slow and triggers the timeout when $P1$ collects the proposed value. Since at most f processes may fail, there are at least f non-faulty processes in addition to $P1$. According to the Algorithm 5 line 94, if these f non-faulty processes determine that there is no failure and do not seek assistance from the DM, they must have received a report from $P1$ indicating no failure. Since $P1$ does produce a report with a failure, then all these f non-faulty processes cannot decide that no failure occurs. If these f non-faulty processes detect failures from the report of $P1$, they have to submit their reports to the DM and request assistance. As a result, at least f processes in addition to $P1$ submit a report and an assistance request to DM. □

Theorem 4. *Termination: Suppose that the RSM of the DM maintains both safety and liveness, then eventually every AP process comes to a decision on the list of values, or crashes.*

Proof. The algorithm begins with each process collecting inputs from other processes. Every non-faulty process is able to receive such inputs from at least $f+1$ processes eventually, and then either receives inputs from or times out on every other process.

Case 1: If a process P_i receives inputs from all processes, then it proceeds with an all-alive report rep_i. Otherwise, the report rep_i indicates which process may have failed by setting the value of that process to *null*. If all reports state all-alive, the process agrees on the list without *null* value as the decision, and the theorem holds.

Case 2: If any report indicates one or more *null* values, then the process P_i requests the decision from the DM by sending its report rep_i. According to Lemma 2, at least $f + 1$ processes have to send the reports. At this point, DM starts to assist in making the decision. After the DM receives reports from these $f + 1$ processes, and either receives reports or times out on every other process, the DM computes a final report and sends it to RSM. Finally, the RSM records the first final report as the decision in each consensus cycle. As long as a decision is made, the DM sends the unified decision to the processes that request assistance. Then, the processes agree on the list of values decided by the DM, and the theorem holds.

Case 3: If a process or set of processes P does receive $2f+1$ reports before the timeout, but another process or set of processes R does not receive $2f+1$ failure-free reports before

timeout, then the process in P must decide in the fast path. However, the processes in R have to send those failure-free reports to the DM. Since no failure occurs according to the received reports, the DM will time out on report collection, and send a failure-free final report to the RSM. The RSM records the decision as no failure then. The DM sends the decision back to the processes that request assistance.

Then the theorem holds. □

Since AP relies on timeout to detect failure, a slow process may trigger timeout and lead the system to a slow path. Under asynchronous mode, a slow process may keep pushing the protocol to a slow path as there is no known upper bound on the message and processing delay. When the system turns to synchronous mode, processes may start the same cycle at very different times. Although the DM in the slow path handles failures, we want to assure that AP runs in the fast path eventually when the system is under synchronous mode if free of failure.

Theorem 5. *Eventual synchronicity: All processes eventually run in fast path if the system runs in the absence of failure and asynchrony starting from a certain cycle.*

Proof. Assume that there is a known upper bound, D, on the message processing and network delay under the synchronous mode. This means that the time to broadcast proposals or reports does not exceed time D. Also, when a process completes or times out on proposals or reports waiting, it takes at most time D to transition to the next step.

When the system turns to synchronous mode, the processes may start the same consensus cycle at a very different time due to the slow path and asynchrony of previous cycles. Assume that cycle x is a such cycle that the system runs in synchronous mode and in the absence of failure from the beginning of the cycle. There may be two cases:

Case 1: no process times out on waiting for proposals according to line 81. Suppose that time t is the time that the last process receives all the proposals. Thus, all the processes should receive $2f + 1$ proposals within the time period $[t - D, t]$. Then, failure-free reports are generated and broadcast within the time period $[t - D, t + D]$. As a result, failure-free reports should be received between time $t - D$ and time $t + 2D$. To enable all the processes to run in a fast path, we just need to set the timeout T greater than $(t+2D)-(t-D) = 3D$.

Case 2: at least one process times out on waiting for proposals according to line 81. Thus, there is at least one failure report which forces all the processes to slow path according to line 94. Then, all processes wait for the decision of cycle x from DM. There are two sub-cases under this situation:

Sub-case 1: all processes request assistance from DM before DM makes a decision of cycle x. Then, once the decision is made, all the processes should receive the decision according to line 100 and line 122. Suppose the first DM node sends the decision back to the processes at time k. Thus, all the processes receive the decision no later than $k + D$. As a result, the processes re-synchronize and start cycle $x + 1$ within the time interval $[k, k + 2D]$. If the timeout T is set greater than $3D$, no process times out on waiting for proposals, which follows case 1 in cycle $x + 1$. As a result, all the processes run the fast path in cycle $x + 1$.

Sub-case 2: At least $f + 1$ fast processes request assistance from the DM before the DM makes a decision of cycle x while at most f slow processes after. As otherwise, DM cannot make a decision according to line 109. Suppose that the maximum time difference between the fast processes and the slow processes regarding requesting assistance is R, and the DM takes time L to process the reports. This means that the fast processes have to wait at least time L to get the decision while the slow processes get the decision without waiting. Thus, the time difference R is shortened by at least time L in cycle $x + 1$. If the processes follow either case 1 or sub-case 1 in cycle $x + 1$, all the processes run the fast path in cycle $x + 1$ or $x + 2$, respectively. If the processes follow sub-case 2 repeatedly, the processes are converged to sub-case 1 due to the shortened time L in each cycle and run in the fast path eventually.

In summary, when $T > 3D$, all the processes eventually run in the fast path. □

4.4 Byzantine Fault-Tolerant Design

In order to tolerate Byzantine failures, we give each process a unique key pair to sign and verify messages. Therefore, the liveness report must include the proposed values associated with the proposer's signature. This is used to prevent a Byzantine process from forging the proposed values of other processes. In addition, we require an acknowledgment signature of each message to eliminate equivocation from a Byzantine process. Equivocation means that a process sends different messages to different processes. There are two types of acknowledgment signatures. One is the value acknowledgment signature which is used to acknowledge the proposed values. The other is the report acknowledgement signature which is used to acknowledge the liveness reports.

Suppose there are $3f + 1$ processes, and each non-faulty process proposes a value. There are at most f processes that may suffer Byzantine failure. Suppose further there are $3m + 1$ DM nodes where at most m nodes may suffer Byzantine failure. We assume that a Byzantine process cannot forge the signature of any other process.

4.4.1 Algorithm

Algorithm 7 is the mechanism under the Byzantine failure model. Compared to the crash-tolerant Algorithm 5, instead of using a single value to indicate the liveness of each process in the report, Algorithm 7 uses the value v_x and the proposer's signature $sig_x(v_x)$ to construct the liveness report rep. Also, we require that the report rep_x of a process P_x include the value acknowledgment signatures, such as line 132. This is used to ensure the integrity of the proposed value, which means that a process P_x sends the same value v_x to every other process. In the pseudo-code, $sig_x(v_j)$ denotes a signature that the value v_j is signed by process P_x. As a result, each report contains up to $3f + 1$ proposers' signatures and $3f$ value acknowledgment signatures. Line 138 records the signatures in a set. In addition, line 146 requires the report acknowledgement signatures $sig(rep_x)$. Since there are a total of $3f + 1$ processes, there should be $3f + 1$ reports at maximum. Therefore, in the normal case, all the signatures in these $3f + 1$ reports are valid, which means that each value in the report is correctly signed by the sender and acknowledged by all the receivers. Moreover, the proposed value $rep.v[j]$ of the process P_j recorded in those $3f + 1$ reports should be the same. Thus, all processes reach agreement on the same list of values.

If any process detects a failure, it must send an assistance request to the DM. We require that the request not only includes its report rep, but also the acknowledgment signatures of the report. An acknowledgment signature of a report rep_x is a signature signed by another process P_i as $sig_i(rep_x)$. These acknowledgment signatures are used to ensure the integrity of the report. This means that the report has been broadcast to other processes already. The procedure is presented as Algorithm 8. Each DM node has to derive a final report $final\_rep$ as follows: if at least one report contains one *non-null* value v_j, then $final\_rep.v[j] = v_j$; otherwise, $final\_rep.v[j] = null$.

The final report also records the signatures included in the assistance request to ensure that a Byzantine DM node cannot forge the final report. Lastly, the final report is sent to the BFT_RSM. The BFT_RSM picks the first valid final report as the DM decision. A valid report represents a report with valid signatures. As long as the DM node receives the decision from the BFT_RSM, it responds to the process that submits the assistance request. The process then determines the list of values based on the decision received.

Compared to the AP under the crash failure model, the AP under the Byzantine failure model employs cryptographic signatures to tolerate Byzantine failures and ensure the integrity of the proposed value as well as the liveness report. Furthermore, the DM uses a BFT RSM instead of a crash fault-tolerant RSM, such as the RSM implemented by PBFT [19] or Mir-BFT [98, 97]. Therefore, on the basis of communication success, we introduce cryptographic signatures to achieve *k-IC*. The Byzantine design requires $4n^2$ messages per

Algorithm 7: Design under Byzantine failure

Procedure *AP_Propose* $P_x(v_x)$
  broadcast $\{v_x\}$
  for *each process* P_i **do**
    request $\{v_i, sig_i(v_x)\}$ from P_i
  end
end
Procedure *AP_Report* $P_x(v_x)$
  wait for $2f + 1$ value and valid signature pairs
  wait for f more pairs until a timeout
  $rep_x.v[x] = v_x$ and $rep_x.sig[x] = \{sig_i(v_x) |\ for\ each\ replied\ process\ P_i\}$
  for *each process* P_i *other than* P_x **do**
    if v_i *received from* P_i **then**
      $rep_x.v[i] = v_i$ and $rep_x.sig[i] = sig_i(v_i)$
    else
      $rep_x.v[i] = null$ and $rep_x.sig[i] = \{\}$
    end
  end
  broadcast rep_x and request at least $2f + 1$ acknowledgement signatures $sig(rep_x)$
end
Procedure *AP_Commit* $P_x(v_x)$
  wait for $2f + 1$ *rep* with valid signatures
  wait for f more until a timeout
  if $3f + 1$ *valid reports were received and they all indicate the same* non-null *value* v_i *for each process* P_i **then**
    for *each process* P_i **do**
      $V[i] = v_i$
    end
  else
    send report rep_x and the acknowledgment signatures of the report to DM
    wait for the decision *Dec*
    for *each process* P_i **do**
      $V[i] = Dec.v[i]$
    end
  end
end

Algorithm 8: DM main loop under Byzantine failure

```
163 Procedure DM P_m
164     wait until the first rep with at least 2f + 1 valid report acknowledgment
          signatures
165     wait for more rep until a timeout occurs
166     if received some valid rep with null value then
167         wait for at least f + 1 reports
168         wait for 2f additional rep received or a timeout occurs
169     end
170     for each process P_j do
171         if at least one rep state rep.v[j] = v_j where v_j ≠ null then
172             final_rep.v[j] = v_j
173             final_rep.sig[j] = sig_j(v_j)
174         else
175             final_rep.v[j] = null
176             final_rep.sig[j] = {rep, sig(rep) | for every received report}
177         end
178     end
179     send the final_rep to BFT_RSM
180     read decision Dec from BFT_RSM
181     send Dec back to the processes who requested assistance
182 end
183 Procedure Execute P_m(final_rep)
184     if Dec is not decided yet and final_rep is valid then
185         Dec = final_rep
186     end
187 end
```

n values in the failure-free case. Due to the requirement of the acknowledgment signatures, the latency is two round trips in the common case, and three round trips plus the processing time of the DM in other cases. In addition to the messages, the Byzantine design requires up to $3n-2$ signatures per liveness report, which result in at most $3n^2-2n$ signatures per consensus cycle. This design uses cryptographic signatures to eliminate the equivocation of the proposed values and the reports. Therefore, the behaviour of a Byzantine process is restricted.

4.4.2 Running Examples

As a running example, consider the case of four processes: $P1$, $P2$, $P3$, $P4$ propose values $v1$, $v2$, $v3$, and $v4$, respectively. We omit the acknowledgment signatures of each report to improve the readability.

Case 1: Suppose that $P1$ receives inputs $v2$, $v3$ and $v4$ from $P2$, $P3$ and $P4$, respectively. Similarly, $P2$, $P3$ and $P4$ receive all the inputs from other processes. The following reports are produced by each process:

$P1 : \{P1 : \{v1, sig_{P1}(v1)\}, P2 : \{v2, sig_{P2}(v2)\}, P3 : \{v3, sig_{P3}(v3)\}, P4 : \{v4, sig_{P4}(v4)\}\}$

$P2 : \{P1 : \{v1, sig_{P1}(v1)\}, P2 : \{v2, sig_{P2}(v2)\}, P3 : \{v3, sig_{P3}(v3)\}, P4 : \{v4, sig_{P4}(v4)\}\}$

$P3 : \{P1 : \{v1, sig_{P1}(v1)\}, P2 : \{v2, sig_{P2}(v2)\}, P3 : \{v3, sig_{P3}(v3)\}, P4 : \{v4, sig_{P4}(v4)\}\}$

$P4 : \{P1 : \{v1, sig_{P1}(v1)\}, P2 : \{v2, sig_{P2}(v2)\}, P3 : \{v3, sig_{P3}(v3)\}, P4 : \{v4, sig_{P4}(v4)\}\}$

If all the above reports send to every process successfully, then every process can decide that the list is $\{v1, v2, v3, v4\}$ without seeking assistance from the DM as line 152 to 154. This is termed DM-unassisted.

Case 2: However, if $P2$ fails with respect to $P3$, the reports become:

$P1 : \{P1 : \{v1, sig_{P1}(v1)\}, P2 : \{v2, sig_{P2}(v2)\}, P3 : \{v3, sig_{P3}(v3)\}, P4 : \{v4, sig_{P4}(v4)\}\}$

$P2 : Failed$

$P3 : \{P1 : \{v1, sig_{P1}(v1)\},\ P2 : \{null\}, P3 : \{v3, sig_{P3}(v3)\},\ P4 : \{v4, sig_{P4}(v4)\}\}$

$P4 : \{P1 : \{v1, sig_{P1}(v1)\}, P2 : \{v2, sig_{P2}(v2)\}, P3 : \{v3, sig_{P3}(v3)\}, P4 : \{v4, sig_{P4}(v4)\}\}$

Since there are only $f+1$ processes, which is 2, that state the same value, all the processes should seek assistance from the DM. And the decision should be $\{v1, v2, v3, v4\}$ according to the report of $P1$ and $P4$ that includes all the values.

Case 3: However, if $P2$ sends different values, $v2'$, $v2''$ and $v2'''$, to different processes, the reports become:

$P1 : \{P1 : \{v1, sig_{P1}(v1)\}, P2 : \{v2', sig_{P2}(v2')\}, P3 : \{v3, sig_{P3}(v3)\}, P4 : \{v4, sig_{P4}(v4)\}\}$

$P2 : Failed$

$P3 : \{P1 : \{v1, sig_{P1}(v1)\}, P2 : \{v2'', sig_{P2}(v2'')\}, P3 : \{v3, sig_{P3}(v3)\}, P4 : \{v4, sig_{P4}(v4)\}\}$

$P4 : \{P1 : \{v1, sig_{P1}(v1)\}, P2 : \{v2''', sig_{P2}(v2''')\}, P3 : \{v3, sig_{P3}(v3)\}, P4 : \{v4, sig_{P4}(v4)\}\}$

Since there are not enough processes that state the same value of $P2$, $P1$, $P3$, and $P4$ observe that $P2$ has failed, then send its report to the DM and wait for the decision at line 156 to 160. Since DM detects three values of $P2$, it sets the value of $P2$ to *null*, the decision process set becomes $P = \{P1,\ P3,\ P4\}$. And the decision values become $\{v1,\ v3,\ v4\}$. $P2$ is suffering Byzantine failure.

Case 4: However, if $P2$ fails with respect to all $P1$, $P3$, and $P4$, the reports become:

$P1 : \{P1 : \{v1, sig_{P1}(v1)\},\ P2 : \{null\}, P3 : \{v3, sig_{P3}(v3)\},\ P4 : \{v4, sig_{P4}(v4)\}\}$

$P2 : Failed$

$P3 : \{P1 : \{v1, sig_{P1}(v1)\},\ P2 : \{null\}, P3 : \{v3, sig_{P3}(v3)\},\ P4 : \{v4, sig_{P4}(v4)\}\}$

$P4 : \{P1 : \{v1, sig_{P1}(v1)\},\ P2 : \{null\}, P3 : \{v3, sig_{P3}(v3)\},\ P4 : \{v4, sig_{P4}(v4)\}\}$

Then, upon receiving the reports, $P1$, $P3$ and $P4$ should send their report to the DM and wait for the decision from the DM as line 156 to 160. As DM detects that three out of four processes report $P2$ fails, the decision set becomes $P = \{P1,\ P3,\ P4\}$. And the decision values become $\{v1,\ v3,\ v4\}$. $P2$ is suffering Byzantine failure and the behaviour is not considered. Cases 2, 3, and 4 are termed as DM-assisted.

4.4.3 Safety and Liveness

This section proves the safety and liveness of the protocol under Byzantine failures. A non-faulty process means a process that does not suffer any failure. A Byzantine process means a process that suffers a Byzantine failure.

Lemma 3. *If every non-faulty process does not receive a proposed value from process P_i, every non-faulty process must decide to seek assistance from the DM.*

Proof. Suppose for contradiction that a non-faulty process P_j makes the decision without assistance from the DM. Then P_j must receive a value proposed by P_i at line 151. This comes to a contradicting observation. □

Lemma 4. *If any non-faulty process receives a* non-null *value from P_i, there is at most one valid such value.*

Proof. Suppose that a process set Q receives v_1 from P_i and another process set R receives v_2 from P_i (line 137), respectively. Q and R are two disjoint, non-empty set of non-faulty processes. Thus, processes in Q should receive at least $2f + 1$ value acknowledgment signatures of v_1, while processes in R should receive at least $2f + 1$ value acknowledgment signatures of v_2. As a result, $4f + 2$ signatures are generated in total for P_i's proposal value. However, there are only $3f + 1$ processes, and only f of them can sign twice because they are Byzantine, which means the maximum number of signatures possible is $4f + 1$, which is less than $4f + 2$. This comes to a contradiction. □

Lemma 5. *If there is a* non-null *value v_x of a process P_x and a valid signature $sig_x(v_x)$ in a report rep_y produced by process P_y, the value must come from process P_x.*

Proof. Suppose a report rep_y that is produced by process P_y includes a *non-null* value v_x of a process P_x and a signature $sig_x(v_x)$. If $P_x = P_y$, then the value is proposed by P_x. If not, since a process cannot forge the signature of any other process, thus if $sig_x(v_x)$ is valid, then the value v_x must come from process P_x. □

Lemma 6. *There is at most one valid report of each process that does not contain* null *value.*

Proof. Suppose process P_i produces two reports rep_{i1} and rep_{i2} that there is no *null* value inside both reports. Suppose rep_{i1} contains value $v1$ and signature $sig_j(v1)$ proposed by P_j, while rep_{i2} contains value $v2$ and signature $sig_j(v2)$ proposed by P_j as well. Since each non-faulty process signs only one value from every other process, and a Byzantine process cannot forge the signature of any other process, it must be that P_i produces two different reports by including the different proposed values of P_i itself, which means $P_i = P_j$. To make the reports valid, P_i should propose $v1$ and $v2$ at line 130 to different processes to collect at least $2f + 1$ signatures on each value at line 132. As a result, according to Lemma 4, it is impossible to have such a situation. □

Lemma 7. *There is exactly one decision made by the DM if any process sends an assistance request to the DM.*

Proof. Similar to the proof under crash failure, if any process submits a request, either the non-faulty process or Byzantine process, the DM can process the reports and send a decision request to the BFT_RSM. If a decision has already been made by the BFT_RSM, no more

decision will be made by the BFT_RSM. If there is no decision made when the BFT_RSM processes the request, then a decision will be made according to the request. □

Lemma 8. *If a non-faulty process sends a valid report with* null *value to the DM, and request assistance, there must be at least* $f + 1$ *non-faulty processes sending their reports to the DM and requesting assistance.*

Proof. According to Algorithm 7 line 146, if a report is valid, it must include at least $2f+1$ acknowledgment signatures. This indicates that at least $2f + 1$ processes receive a report with a *null* value. According to the algorithm, if a process receives a report with a *null* value, then it cannot decide the outcome without seeking assistance from the DM. Since there are at most f Byzantine processes, as a result, at least $f + 1$ non-faulty processes send their reports to the DM and request assistance. □

Theorem 6. *Validity: If a non-faulty process decides a* non-null *value* v_i *in* V *and process* P_i *is not faulty, then* v_i *must be proposed by process* P_i*.*

Proof. There are two cases that a non-faulty process decides the list of values V.

Case 1: No failure and no timeout occurs as line 152, all reports state the same *non-null* value v_i for process P_i.

According to the algorithm, all the processes have to choose v_i as the decided value of P_i in V.

Case 2: The decision V is made by the DM.

According to Lemma 7, there is only one decision V. Since P_i is non-faulty, all the reports must contain either *null* or v_i as the proposed value of P_i. According to the decision process of DM, Algorithm 7 line 170 to line 178, v_i must be chosen as the decided value of P_i in V if no *null* value in any report. Otherwise, the AP has to wait for at least $f + 1$ reports. If there is at least one report with v_i of P_i from a non-faulty process, then, the AP has to use v_i as the proposed value of P_i in V. If all received reports record *null* of P_i, the decided value of P_i can only be *null*.

As a result, in both cases, the validity was proved. □

Theorem 7. *Agreement: No pair of non-faulty processes ever decides different* V*.*

Proof. It is similar to the proof under crash failure except that the property merely restricts non-faulty processes since the Byzantine process can behave arbitrarily.

If any non-faulty process decides the outcome of the list of values, the outcome must belong to exactly one of the following two cases:

Case 1: The process receives $3f + 1$ valid reports that every report includes the same *non-null* value and a valid signature of each process, then the outcome is the whole list of the values without seeking assistance from the DM.

Case 2: The process either receives a report includes *null* value, or does not receive $3f + 1$ reports before the timeout, then the process should send an assistance request to the DM, and decide the outcome according to the decision made by the DM.

Thus, if the outcomes of any two non-faulty processes belong to case 1, then the outcome will be the whole list of values. According to Lemma 3 and 4, each non-faulty process must only propose exactly one value in case 1. Thus, there is one unique list of values that contains all the proposed values from each process.

If the outcomes of any two non-faulty processes belong to case 2, due to Lemma 7, the outcomes of these two processes are the same as well.

Now, consider the last situation, some non-faulty processes decide the outcomes as case 1, and some non-faulty processes decide the outcomes as case 2. Suppose, processes in the non-faulty process set Q decide that no need to seek assistance from the DM, while processes in the non-faulty process set R decide to seek assistance from the DM. Q and R are disjoint, non-empty non-faulty processes set. According to the algorithm, processes in Q must have received $3f + 1$ valid reports without a *null* value of any process. This means that the reports produced by the non-faulty processes in R are all valid and contain *non-null* value only. However, a Byzantine process in R can send the same valid report as it sends to the processes in Q, or a valid report with *null* values of some processes, or nothing, to the other processes in R. Due to Lemma 6, the Byzantine process cannot send a different valid report without a *null* value. Thus, the non-faulty processes in R either miss some reports or received some reports with a *null* value. But the report of the non-faulty process itself must be valid and not include any *null* value. To guarantee safety, the DM must also decide the same list of values as the processes in Q. This leads to two sub-cases.

Sub-case 1: The DM only receives valid reports without a *null* value until timeout in line 165.

Sub-case 2: The DM receives some valid reports with a *null* value before timeout in line 168. The DM should wait for at least $f + 1$ reports before making the decision.

Under sub-case 1, according to the analysis above, the reports must be the same as the non-faulty processes in Q received. Thus, the DM must decide the same list of values as

the processes in Q decided. As a result, the non-faulty processes in R receive the same list of values as the non-faulty processes decided in Q.

Consider sub-case 2, some reports with one or more *null* values, termed as Byzantine reports here. As the DM requires that each report should include at least $2f + 1$ valid acknowledgment signatures from other processes, and there are at most f Byzantine processes, at least $f + 1$ non-faulty processes receive a Byzantine report or receives less than $3f + 1$ reports. As a result, at least $f + 1$ processes seek assistance from the DM. The reports from these $f + 1$ non-faulty processes in R are valid without a *null* value. Otherwise, the processes in Q cannot proceed to line 152. Since the non-faulty processes in R seek assistance with valid reports without *null* value, the DM should receive at least $f + 1$ valid report without a *null* value before making the decision. Then according to line 170 to line 178, the decision of the DM must be the same as the valid report, which is also the same as the outcome of the non-faulty processes in Q made. Thus, the non-faulty processes in R can receive the same value set as the processes in Q. Then the theorem holds. □

Theorem 8. *k-Completeness: If the protocol can tolerate up to f Byzantine failures out of total $3f + 1$ processes, then the number of* null *values in V is no greater than 0 if there is no failure and asynchrony. Otherwise, the number of* null *values is no greater than $2f$.*

Proof. Similar to Validity, there are two cases that a non-faulty process decides the list of values V.

Case 1: No failure and no timeout occurs. All reports state the same *non-null* value v_i for process P_i of all the processes. As a result, no *null* value in V, which is 0-Completeness.

Case 2: The decision V is made by the DM.

The decision of the DM is according to the reports submitted by each process, and if any report includes a *non-null* v_i of P_i, then the decision value of P_i can be either v_i if at least one report state v_i, or *null* otherwise. Since each report includes at least $2f + 1$ *non-null* values out of total $3f + 1$ values, and there are at most f Byzantine values within these $2f + 1$ values. As a result, the total number of *null* values cannot exceed f, which is f-Completeness. □

Theorem 9. *Termination: Suppose that the DM maintains safety and liveness, then eventually every non-faulty process comes to a decision of the list values.*

Proof. The algorithm begins with each process collecting inputs, a value and a signature pair, from all the other processes. Every non-faulty process is able to receive such inputs from at least $2f + 1$ processes eventually, and then either receives inputs from or times

out on every other process. If the process receives inputs from all the processes then it proceeds with a report *rep* that includes a value and signature pair. Otherwise, the report *rep* indicates which process may fail by setting the value of that process as *null*. Then each process proceeds to collect reports, at least $2f + 1$ reports. If all $3f + 1$ reports state the same value for each process, then the process agrees on the whole set of values as the decision, and the theorem holds. On the other hand, if there do not exist enough reports stating the same value of a process, or receive less than $3f + 1$ reports until the timeout, then the process requests assistance from the DM by sending its report *rep* to the DM. This comes to two different cases:

Case 1: All non-faulty processes seek assistance from the DM.

Case 2: Only some of the non-faulty processes, equal to or greater than one but not all, seek assistance from the DM.

Considering case 1, all non-faulty processes send their reports to the DM and request assistance. Since there are at most f Byzantine processes, the DM can receive at least $f + 1$ valid reports from non-faulty processes. Thus a DM node can proceed to derive a final report and send the report to the BFT_RSM.

Under case 2, there must be at least one non-faulty process making a decision without the help of the DM. Then when a process sends its report to the DM, if the process is non-faulty, it must send a valid report without a *null* value. This is because there is at least one process that receives all the failure-free reports and does not seek assistance from the DM. If the process is a Byzantine process, according to Lemma 4 and 6, it can only either send a valid report without *null* value as it sends to every other process, or send a valid report with a *null* value. Thus, if the DM only receives valid reports without a *null* value, each DM node does not need to wait for enough reports and can proceed to make the decision (line 166). If the DM receives some valid reports with *null* values, because of Lemma 8, the DM can expect at least $f + 1$ reports from different processes. Thus, each DM node can proceed to make the decision. As long as the decision is made, the DM sends the decision back to the processes that request assistance. Then the non-faulty processes can agree with the list of values decided by the DM, and the theorem holds. □

Theorem 10. *Eventual synchronicity: All processes eventually run in the fast path if the system turns to synchronous mode and is free of failure starting from a certain cycle.*

Proof. Similar to crash fault-tolerant design, assume that there is a known upper bound, D, on the message processing and network delay under the synchronous mode. This means that the time to broadcast proposals or reports does not exceed D. Also, when a process

complete or timeout on proposals or reports waiting, it takes at most time D to transition to the next step.

When the system turns to synchronous mode, the processes may start the same consensus cycle at a very different time due to the slow path and asynchrony of previous cycles. Assume that cycle x is a such cycle that the system runs in synchronous mode and in the absence of failure from the beginning of the cycle. There may be two cases:

Case 1: no process times out on waiting for proposals according to line 137. Suppose that time t is the time that the last process receives all the proposals and signature pairs. Thus, all the processes should receive $3f+1$ valid proposals within the time period $[t-D, t]$. Then, failure-free reports are generated within the time $t-D, t+D$ and broadcast within the time period $[t-D, t+2D]$. As a result, failure-free reports should be received between time $t-D$ and time $t+3D$. To enable all the processes run in the fast path, we just need to set the timeout T greater than $(t+3D)-(t-D)=4D$.

Case 2: at least one process times out on waiting for proposals according to line 137. Thus, there is at least one failure report which forces all the processes to slow path according to line 151. Then, all processes wait for the decision of cycle x from DM. There are two sub-cases under this situation:

Sub-case 1: all processes request assistance from DM before DM makes a decision of cycle x. Then, once the decision is made, all the processes should receive the decision according to line 157 and line 181. Suppose the first DM node sends the decision back to the processes at time k. Thus, all the processes receive the decision no later than $k+D$. As a result, the processes re-synchronize and start cycle $x+1$ within the time interval $[k, k+2D]$. If the timeout T is set greater than $3D$, no process timeout on waiting for proposals which follow case 1 in cycle $x+1$. As a result, all the processes run the fast path in cycle $x+1$.

Sub-case 2: at least $2f+1$ fast processes request assistance from the DM before the DM makes a decision of cycle x while at most f slow processes after. Otherwise, DM cannot make a decision according to line 166. Suppose that the maximum time difference between the fast processes and the slow processes regarding requesting assistance is R. And the DM takes time L to process the reports. This means that the fast processes have to wait at least time L to get the decision while the slow processes get the decision without waiting. Thus, the time difference R is shortened by at least time L in cycle $x+1$. If the processes follow either case 1 or sub-case 1 in cycle $x+1$, all the processes run the fast path in cycle $x+1$ or $x+2$, respectively. If the processes follow sub-case 2 repeatedly, the processes are converged to either case 1 or sub-case 1 due to the shortened time L in each cycle and run in the fast path eventually.

In summary, when $T > 4D$, all the processes eventually run in a fast path. □

4.5 Implementation

We implement prototypes of Antipaxos under the crash failure model (APCFT) and under the Byzantine failure model (APBFT) in Golang, respectively.

4.5.1 Crash Fault-Tolerant Implementation

In APCFT implementation, we use it to build a replicated key-value store. Each process runs APCFT and executes requests in cycles. To optimize the throughput, the processes reach agreement on an ordered list of batched requests in each cycle. The cycle number is increasing monotonically. Since there is no guarantee that all the processes run the same cycle and process the requests at the same speed, we introduce pipelined cycles to reduce the impact of asynchrony. Normally, a new cycle starts if the request batching time is up. But to offset the impact of asynchrony, if a process receives a proposal of a cycle whose cycle number is greater than the max cycle number it is running, then the process has to start a new cycle immediately even if there is no client request. The stages of the AP can be pipelined to allow multiple cycles to run concurrently, as shown in Figure 4.2. The algorithm is divided into 5 steps. The first step is to collect and batch client requests into a proposal. Then each replica broadcasts its proposal associated with a unique cycle number to every other replica. Next, every replica either receives all the proposals of a cycle or a timeout occurs of that cycle. A report is produced consequently to indicate whether the proposals are successfully received from every other replica. Then, the report should be broadcast to other replicas. Next, the cycle is committed according to the reports. In the end, each replica can execute the committed requests in each cycle with a predetermined order, e.g., in round-robin order. Since the execution is based on the cycle numbers only, each consensus cycle is independent and can be run concurrently. The design of pipelined cycles also makes the APCFT efficient in a WAN.

Additionally, to reduce the size of a report, we require each report to include only the liveness status instead of the original requests. As a result, a report uses a Boolean value to indicate whether a process is alive or may have failed. If there is a report for each process and there are no failures in these reports, the processes proceed in the fast path. Otherwise, the processes have to retrieve the lost requests by querying other non-faulty processes.

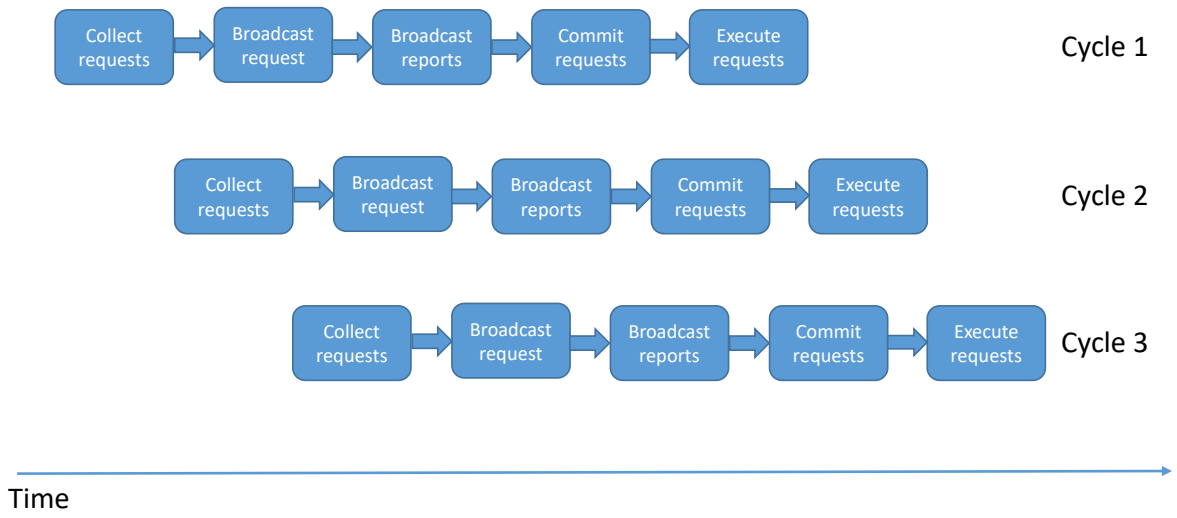


Figure 4.2: Pipelined Antipaxos

Failure handling is a key feature of a consensus protocol. When a process fails, APCFT reacts by contacting the DM for assistance in each affected consensus cycle. DM is implemented using EPaxos. Although this prevents the protocol from stalling, it also leads to high latency. To reduce this overhead, when a process is unavailable for many consecutive cycles, we automatically exclude the unresponsive process from the computation for a number of additional cycles, which amortizes the disruption over many cycles. As a last resort, a process that is suspected to have failed permanently can be removed from the protocol by updating the group membership, with the option of rejoining later under a new identity. The membership reconfiguration is out of the scope of this paper. In another case, when a process merely runs slow, other processes suspect that the process has failed and seek assistance from the DM. In other consensus protocols, such as Paxos [55] and PBFT [19], if the slow process is the leader, they have to run a view-change protocol to select a new leader after a timeout. If the slow process is not the leader, prior protocols either treat the slow process as a failure and ignore it, or halt and keep waiting for a reply. In our implementation, a slow process should catch up because of the design of pipelined cycles. However, if it still falls behind, the DM treats the slow process as a failure and

handles it with the above failure procedure. This essentially avoids the asynchrony that keeps pushing the consensus to the slow path. We examine different strategies for dealing with failed processes in the next section.

4.5.2 Byzantine Fault-Tolerant Implementation

Similar to APCFT implementation, APBFT enables request batching and pipelined cycles to improve performance. However, instead of a replicated key-value store, APBFT uses a typical workload consisting of 500-byte requests, which is approximately the average size of a Bitcoin transaction [103]. Since the report of APBFT has to carry signatures for fault tolerance, each process is uniquely identified by an ECDSA [46] key pair. The private key is used to sign the proposals and reports while the public key is used to verify the signatures. To reduce the size of a report, we require each report to include only the SHA256 [71] hash of the original requests. Two special hashes are used to indicate empty requests and processes may have failed, respectively. If no failure occurs, the processes proceed on the fast path. Otherwise, the processes have to seek assistance from the DM and retrieve the missing requests that match the hash in the decision from other non-faulty processes.

DM under APBFT is implemented using PBFT [19]. We leverage the implementation of MirBFT [98] to emulate PBFT. We use the same approach as APCFT to handle faulty processes. When a failure first occurs, the processes contact DM for help to solve the failures. If the failure occurs for a number of additional and consecutive cycles, APBFT excludes and ignores the failed processes from the consensus. This is to ensure that most cycles proceed on the fast path. Our experiments evaluate different strategies for fault handling.

Latency (ms)	USE	SAE	EUC	APN	USW	APS	CAC
USE	-						
SAE	123	-					
EUC	99	200	-				
APN	156	251	267	-			
USW	48	170	140	101	-		
APS	198	303	130	124	212	-	
CAC	24	124	90	141	63	188	-

Table 4.1: Round-trip network latency (ms)

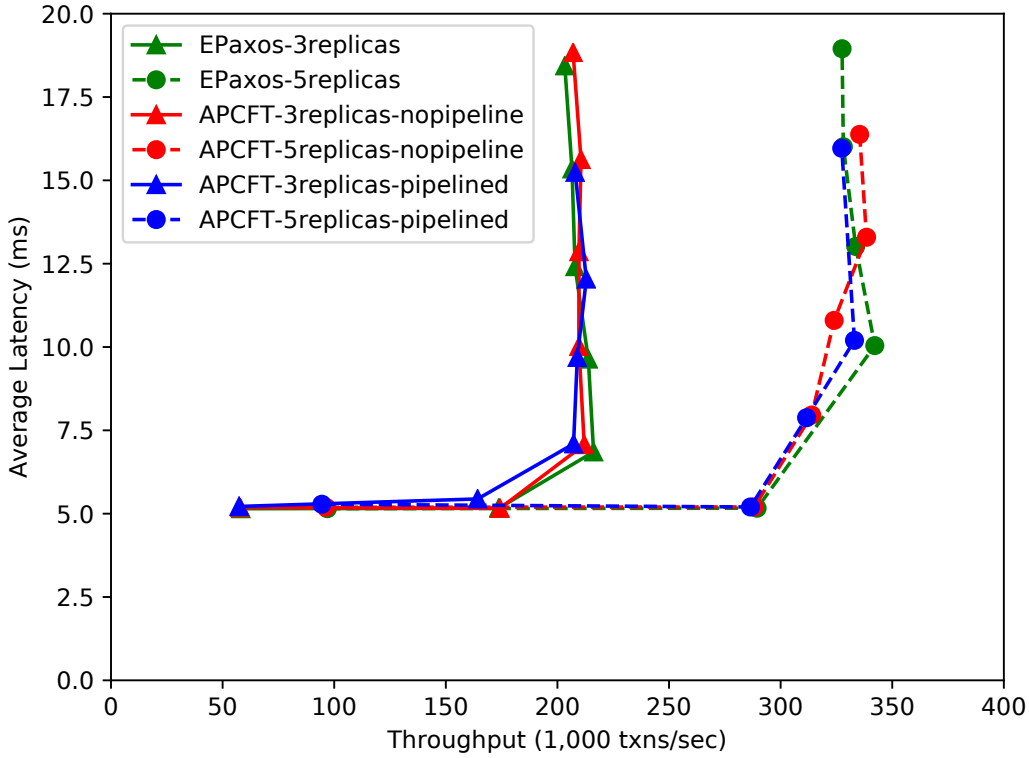


Figure 4.3: APCFT Throughput-latency in a single datacenter.

4.6 Evaluation

We evaluate our prototype on Amazon EC2. Each process runs in a c4.xlarge instance (4 cores, 2.9GHz Intel Xeon E5-2666 v3 processor, 7.5GB memory). Each instance runs Amazon Linux 2 and uses Golang version 1.13.6. Each process runs on a dedicated instance.

To evaluate the performance, we deploy the AP system in both a single datacenter and multiple datacenters. A client process runs multiple threads to imitate different users sending requests concurrently. Each user sends only one request at a time and sends requests sequentially, which means closed-loop users. The throughput is controlled by increasing the number of concurrent users. The execution waiting time of each request is the time from when the user sends the request to the process to when the user receives

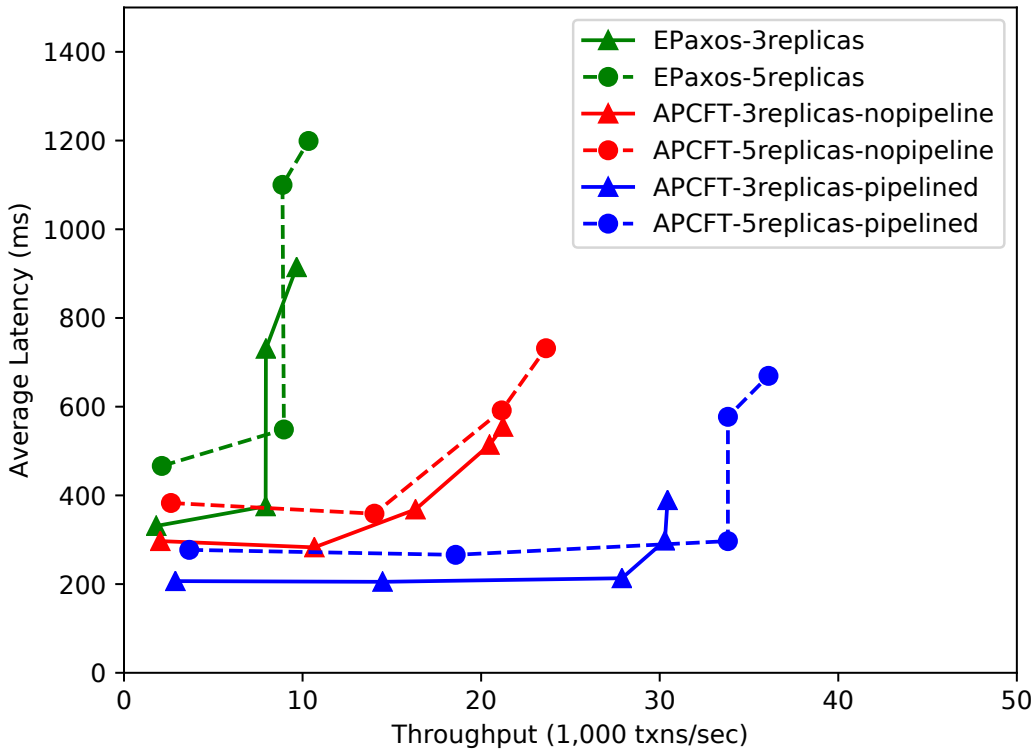


Figure 4.4: APCFT Throughput-latency in multiple datacenters.

the reply from the process. The process only replies after executing the request. We use two different settings to measure the performance of AP, one using pipelined cycles, and the other without pipelining. We run each experiment for a total of 30s, but only measure the throughput and latency in the middle 20s to capture the steady-state performance. We calculate the throughput as the number of transactions per second (txns/sec) and the average latency of all executed requests in the 20s interval.

4.6.1 APCFT Experiments

Under the CFT setting, we compare against a high-performance and open-source protocol, EPaxos. The workload is a 17-Byte key-value write command (1 Byte write operation, 8

Bytes key, and 8 Bytes value). Also, each process batches up to 1000 commands every 5ms.

In the first experiment, we deploy the system in a datacenter located in US-EAST (Ohio). This configuration minimizes the impact of network delays. We measure the throughput and latency of 3 replicas and 5 replicas, respectively. Figure 4.3 shows the results of this single datacenter experiment. In a low latency network, AP achieves similar peak throughput and latency to EPaxos. Compared to 3 replicas, 5 replicas provide better peak throughput. Thus, the system can scale according to the number of replicas in a low-latency network. This is because AP and EPaxos have the same message complexity, which is $O(n)$ per consensus cycle where n is the number of replicas. Noteworthy, the pipelined design in the low-latency network does not have an appreciable impact on the performance. This is because the time consumed in each cycle is very short.

The second experiment is deployed in a wide-area environment. We choose 5 different AWS EC2 datacenters in the following regions: US-east2(USE), SA-east1(SAE), EU-central1(EUC), AP-northeast1(APN) AND US-west2(USW). The network latency between these datacenters is listed in Table 4.1, and the average latency is about 150ms. The 3 replicas experiment uses the first three datacenters, and the 5 replicas experiment uses all five datacenters. Figure 4.4 shows that even without pipelining, AP achieves 2x the peak throughput compared to EPaxos. This is because of the lower execution latency, as shown in the figure. In the common case, AP only requires one round-trip. But, EPaxos requires one and a half round-trips in its fast path, and more round-trips plus dependency check in its slow path. When the consensus cycle is pipelined, the peak throughput increases to 3x that of EPaxos. This is because the pipelined implementation efficiently uses the network bandwidth without having to wait for a reply from the previous cycle in the WAN.

In the last experiment, we measure the performance in a failure case. Our DM system comprises 3 replicas. Each DM node runs in an EC2 instance. We deploy 3 replicas and one client in the US-EAST region. The DM system is also deployed in the same region. In order to have a stable sending rate, the client sends requests in an open loop to a non-faulty replica, issuing new commands at 10,000 requests per second without waiting for the replies of the old commands over the testing period. We measure the execution latency of each command and calculate the throughput every 0.2s. One of the replicas crashed after about 5s into the experiment. We implement three strategies to deal with the failed replica. The first one (S1) is that when the system detects a failure, it immediately excludes the replica from the computation. The timeout of each step is set to a small value, which is 10ms. The second strategy (S2) is to exclude the failed replica if the system detects a failure for multiple (100 in this setting) consecutive cycles. The timeout of each step is still 10ms. In the last strategy (S3), the system excludes the failed replica immediately if a failure is

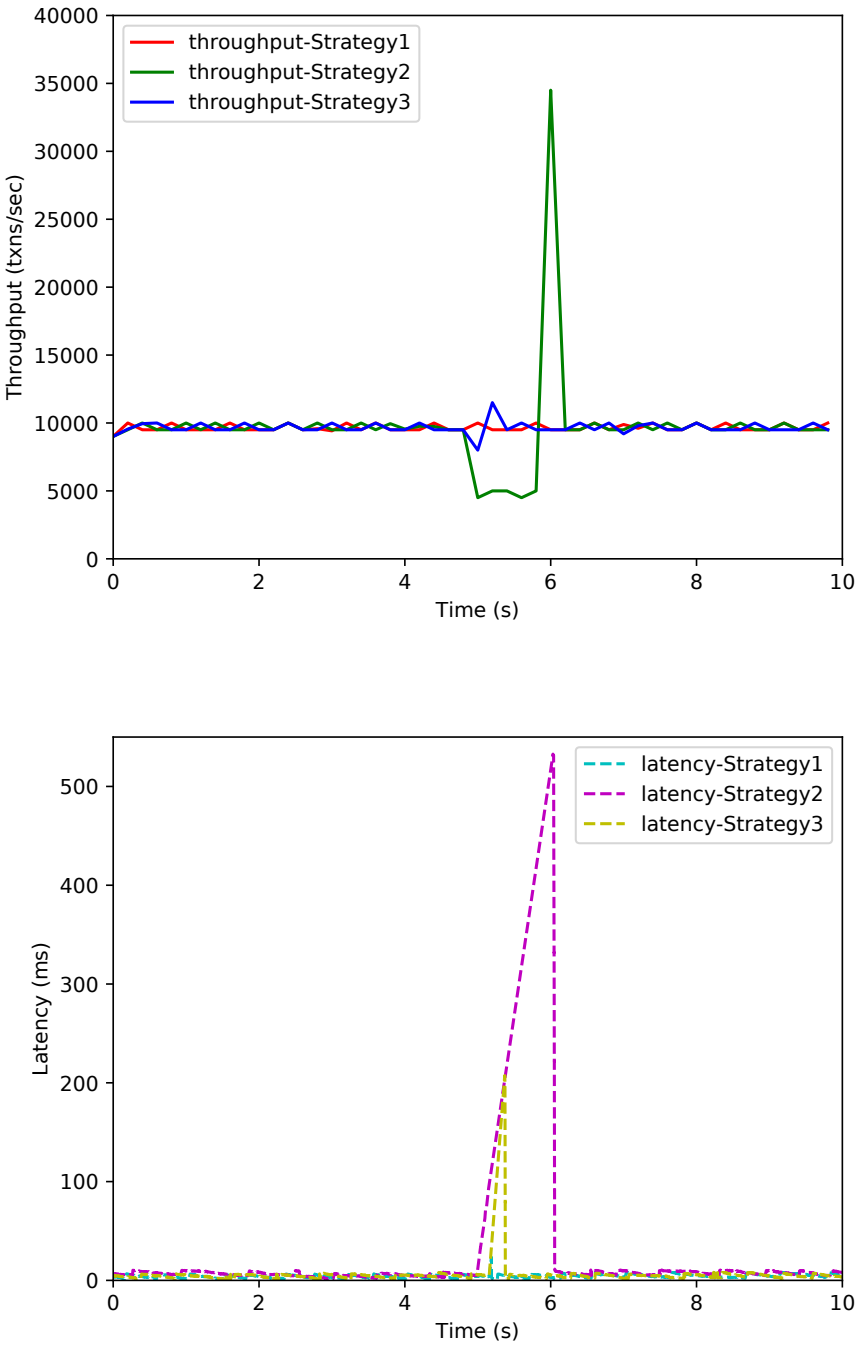


Figure 4.5: APCFT fault recovery experiment, one replica is killed after 5s into the experiment.

Max batch size	2MB
Max batches ephemeral epoch	256
Bucket rotation period	256
Buckets per leader	2
Checkpoint period	128
Watermark window size	256

Table 4.2: MirBFT configuration

detected. But the timeout is set to a larger value, 100ms.

Figure 4.5 shows the results of three different strategies. We can see that S1 almost handles the failure without interrupting system performance. S2 reduces throughput by about half and significantly increases latency. Finally, for S3, there is a noticeable fluctuation in both the throughput and latency. The design of these three strategies is related to the likelihood of false-positive failure. This is because a slow replica may push the system to the slow path. Longer timeout, more consecutive failure cycles, or both, resulting in a lower likelihood of false-positive failure detection. But the trade-off is performance degradation. Although S1 gives the best performance in handling the failure, a short timeout may keep excluding non-faulty but slow processes from the computation, which jeopardizes the system fault tolerance. S2 shows us that continuously running at a slow path has a significant impact on performance. To best use AP, we recommend S3, which has a reasonable timeout compared to the network latency. In summary, if we can estimate the upper bound of the message and process delay, the system runs in the fast path most of the time, which provides high performance. If a timeout occurs, either due to process failure or asynchrony, the DM temporarily excludes the process from the computation. When a process is recognized as a failure multiple times in succession, DM permanently removes it from the system. When the failed replica recovers, it has to either catch up with the computation or rejoin the execution by applying for a system reconfiguration.

4.6.2 APBFT Experiments

Under the BFT setting, we compare against MirBFT [98], a multi-leader BFT protocol implemented in Golang. The workload is a 500-Byte request. Each process batches up to 2MB requests.

The first experiment is conducted in a single datacenter, US-EAST (Ohio). Each cycle

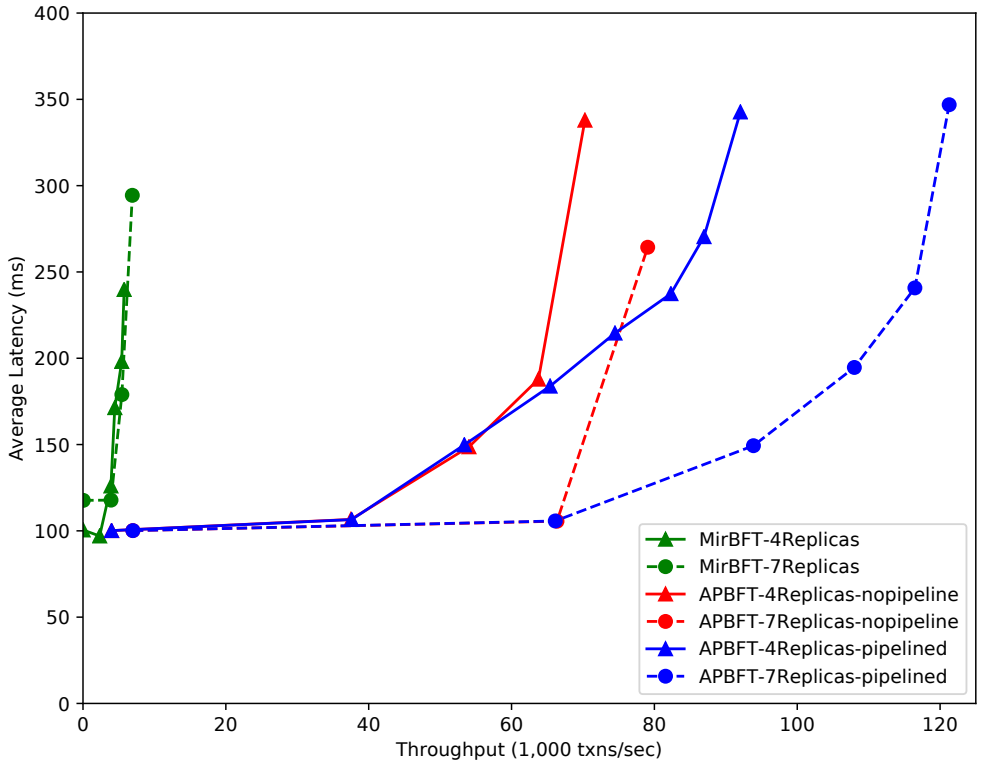


Figure 4.6: APBFT Throughput-latency in single datacenter.

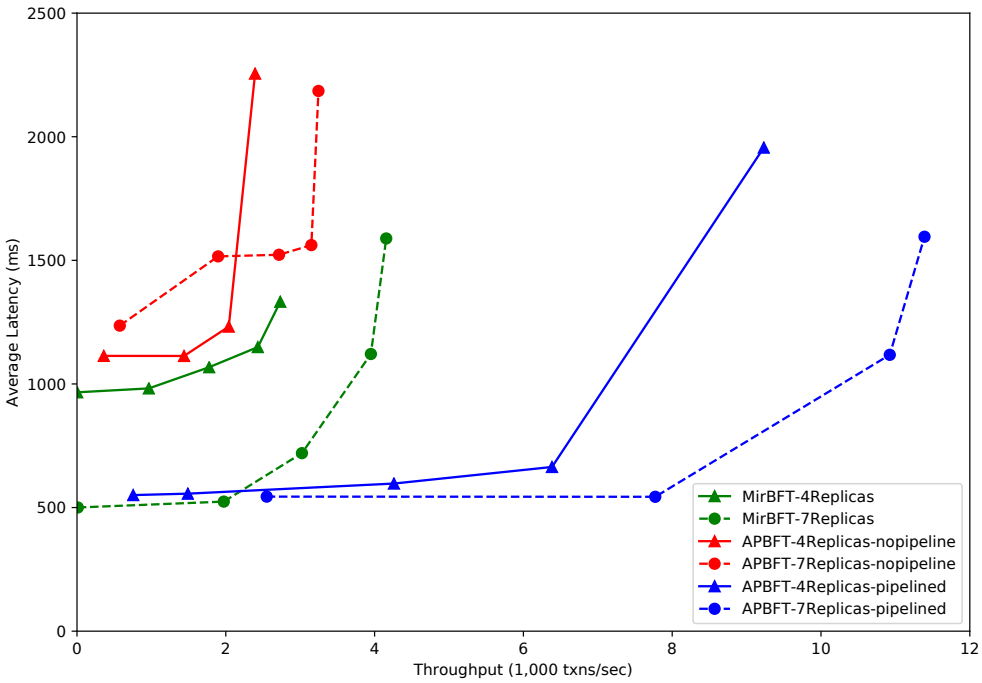


Figure 4.7: APBFT Throughput-latency in multiple datacenters.

batches 100ms in both APBFT and MirBFT. Other parameters of MirBFT follows Table 4.2 which is proposed in [98]. Under this setting, the network latency is minimized. We measure the throughput and latency of 4 replicas and 7 replicas, respectively. The results are shown in Figure 4.6. When the network latency is low, APBFT shows much higher peak throughput than MirBFT. This is because MirBFT multiplex PBFT to achieve multiple instances of consensus concurrently while APBFT merely requires two rounds of all-to-all messages in its fast path. Regarding the scalability, 7 replicas show better peak throughput. In addition, due to the signature verification, pipelined design in the low-latency network shows higher peak throughput.

The second experiment is deployed in multiple geo-distributed datacenters, which are US-east2(USE), SA-east1(SAE), EU-central1(EUC), AP-northeast1(APN), US-west2(USW), AP-south (APS), and CA-central(CAC). Each cycle batches 200ms in both APBFT and MirBFT. The network latency between these dataecnters is shown in Table 4.1, and the

average latency is about 150ms. The 4 replicas experiment is deployed in the first four datacenters, and the 7 replicas experiment is deployed in all sever datacenters. The results are shown in Figure 4.7. Without pipelining, APBFT is slightly inferior to MirBFT. This is because that APBFT is dominated by the slowest replica due to the all-to-all communication whereas MirBFT relies on PBFT which only requires a quorum. However, when we enable pipelining, APBFT outperforms MirBFT by more than 3 times and 2 times in 4 and 7 replicas, respectively. The pipelined implementation process proposals and reports efficiently without waiting for the previous cycle in the WAN.

The last experiment is to measure the performance in a failure case. Both APBFT replicas and DM nodes are deployed in the same datacenter in the US-EAST region. We deploy 4 APBFT replicas, 4 DM nodes, and 1 client. Similar to the APCFT fault tolerance experiment, we use an open-loop client to send requests at a constant speed, 10,000 requests per second, to a non-fault APBFT replica. We measure the execution latency of each request and calculate the throughput every 0.2s. We simulate a replica crash by killing a replica after about 5s into the experiments. Three different strategies are applied to handle the failure. Strategy 1 (S1) excludes the failed replica from the computation if DM detects a single failure in any consensus cycle. The timeout of each step is set to 100ms. Strategy 2 (S2) excludes the failed replica if a failure occurs for a consecutive 20 cycles. The timeout of each step is still 100ms. The last strategy, strategy 3 (S3), excludes the failed replica immediately if a failure is detected as strategy 1, but the timeout is set to a larger value, 200ms. We would like to use these different strategies to show the sensitivity of APBFT with a different configuration.

We show the results in Figure 4.8. There is a fluctuation in all three strategies when a failure occurs. The throughput decreases and the latency increase significantly when a failure occurs at about 5s, but the system is still alive. This throughput decrease and latency increase are due to the mechanism of DM. When an AP replica detects a failure, it has to report to DM and waits for the decision from the DM. The requests cannot be executed before the decision is received. S1 takes the least time to recover to normal as it excludes the failure replica from the computation immediately when the failure is detected. S2 takes a longer time to recover due to multiple assistance cycles. S3 performs better than S2 but worse than S1 which is because of a longer timeout. Similar to APCFT, the design of failure handing depends on the likelihood of false-positive failure which is because of the slow replica. For practical applications, to make AP runs in the fast path in most cycles, a good approach is to measure and estimate the upper bound of the message and process delay. If a timeout occurs, either due to process failure or asynchrony, the system excludes the process from the computation. When a process is recognized as a failure multiple times in succession, DM permanently removes it from the system. The failed replica has

to either catch up with the computation or rejoin the execution by applying for a system reconfiguration after recovery.

4.7 Related Work

Paxos [55] is a well-known crash-tolerant consensus protocol. A process is selected as the proposer to lead the consensus protocol and communicates with a majority quorum of acceptors. The acceptors are the processes to maintain fault-tolerant memory. The learners are the processes to execute the requests. A process can take any or all the roles in the protocol. At the beginning of the protocol, a leader sends a message to all the acceptors in the first phase and waits for enough reply messages. Then the leader sends the accept message to all the acceptors which include the proposed value in phase two. Then all the correct acceptors accept the proposed value and broadcast the success message to both the leader and all the learners to finish the consensus. Paxos can guarantee safety. Liveness is conditioned on that there are at most f acceptors out of total $2f + 1$ acceptors that may suffer crash failure, and a message cannot be infinitely delayed. Besides, if multiple leaders are competing with each other, the protocol may also be stuck.

Due to the reliance on a single proposer, the performance of Paxos is limited. Many variants of Paxos [88] have been proposed to improve the performance. For example, Fast Paxos [56] reduces the agreement time by having proposers send directly to an increased number of acceptors. Mencius [9] increases the throughput by rotating the leaders across different consensus cycles. Besides Paxos family, Raft [81] has similar fault-tolerance and performance compared to Paxos. But it is designed to be more understandable than Paxos. Zab [47] is an atomic broadcast protocol employed by Zookeeper [43]

EPaxos [75] is an improvement on Paxos where every replica handles requests from the clients. When a replica receives a request from a client, the replica becomes the request leader and forwards the request to at least a fast-path quorum of replicas. If the request leader receives enough replies where all the replies have the same attributes. Then, the request can be committed in a fast path. Otherwise, the request has to commit through a slow path which is based on Paxos. If the request leader commits a request, it simply broadcasts this request to everyone else to execute. In the fast path, EPaxos only requires one round-trip communication latency to commit a non-interfering request. EPaxos also employs batching techniques to increase throughput. However, each EPaxos replica runs consensus one cycle at a time, which limits its throughput in WANs. Furthermore, concurrent conflicting commands, which are common when commands are batched, always fall to the slow path. Compared to EPaxos, Altas [31] improves the performance by using a

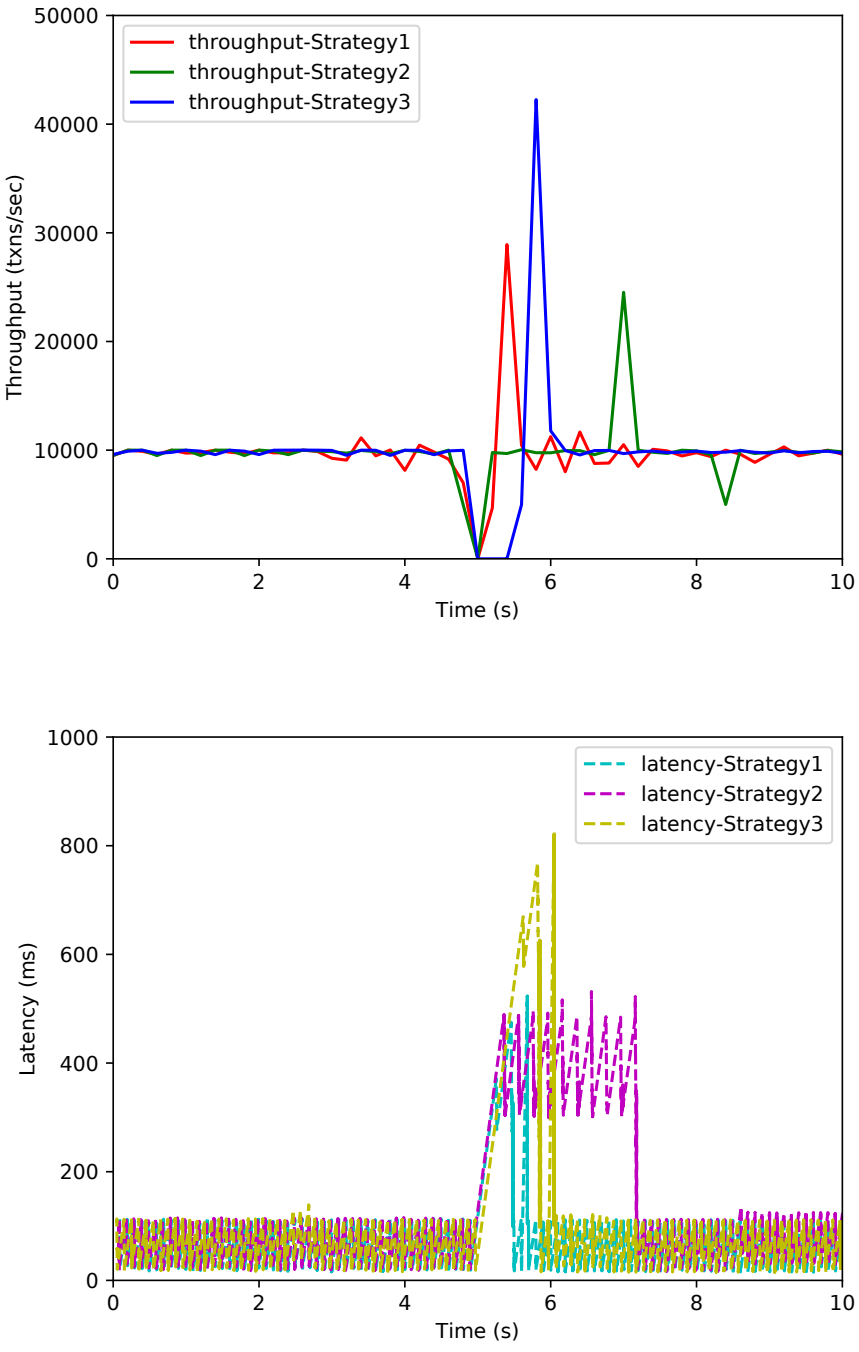


Figure 4.8: APBFT fault recovery experiment, one replica is killed after 5s into the experiment.

minimized quorum for its fast path. Canopus [89] is a consensus protocol with a hierarchical tree architecture. The top layer of this hierarchy uses an efficient all-to-all broadcast, similar to Antipaxos, which allows Canopus to outperform EPaxos in terms of throughput for read-heavy workloads. Nevertheless, Canopus lacks the ability to tolerate a network partition or failures of an entire subtree of the hierarchy.

Apart from crash fault-tolerant consensus protocols, some other protocols have been proposed to tolerate Byzantine failures [57]. PBFT [19] is the most widely used single-leader Byzantine fault-tolerant consensus protocol as we introduced in 2. FastBFT [59] proposes a novel message aggregation technique based on hardware-based trusted execution environments to reduce message complexity. Gosig [58] supports multiple proposers, but only one of the proposals can be selected in each cycle. There are a number of other single-leader BFT protocols such as SBFT [39], HotStuff [107], Zyzzyva [52], Aliph[38], etc.

Considering the leaderless and the multi-leader approaches, set-union consensus [28] proposes that a set of processes reach agreement on a set of values instead of a single value in a consensus cycle, which is similar to k-*IC*. Set-union consensus requires that there is no conflict or invalid element in the decision set. The definition of a conflict or invalid is application-specific. Also, pipelining was not considered in the set-union consensus. There are some other variants of leaderless approach consensus protocols, such as Honey Badger [74], DBFT [21], Red Belly [23], etc.

Mir-BFT [98] is a multi-leader approach consensus protocol where each leader runs an instance of PBFT. It splits the consensus into epochs. In each epoch, each leader is assigned a fixed number of buckets. Requests are matching to the buckets using their hash. Buckets are rotated across different epochs. Finally, requests are interleaved in a unique sequence according to the order of the buckets. MirBFT can emulate PBFT by setting only one leader and making every epoch stable. However, there is no mechanism for a slow leader to catch up. As a result, the slow leader dominates the performance. In addition, MirBFT requires $O(n^3)$ message complexity if all the nodes are leaders.

In addition, Steward [3] and RCanopus [48] are Byzantine consensus protocols that improve performance by using a hierarchical architecture. They comprise multiple sites, each using Byzantine consensus internally to reach local decisions and certify them. Agreement across sites is achieved using a Paxos-like single-leader consensus protocol in Steward. In contrast, RCanopus uses an all-to-all broadcast to synchronize sites, similar to Canopus [89]. Fault tolerance is limited, as in Canopus, since the protocol stalls when a site becomes unavailable. AP is an elaboration of the convergence module (CM) in RCanopus [48]. The CM is introduced to make RCanopus resilient against network partitions and site failures.

4.8 Conclusion

In this paper, we first formalize the *k-IC* problem. Then we propose Antipaxos, a scalable leaderless protocol, to solve this problem. In the absence of failures and asynchrony, the system bypasses classical consensus protocols and reach agreement efficiently using all-to-all broadcast. Otherwise, the algorithm still maintains safety and liveness with the help of the DM. It achieves this by sending liveness reports and using communication success as a failure detector in the crash failure model and adds cryptographic signatures to tolerate Byzantine failures. In addition, proposal batching and pipelined consensus cycle allow AP to reduce the impact of asynchrony and achieve high scalability in WANs. We implement system prototypes under crash failure and Byzantine failure. And we conduct experiments on Amazon to demonstrate the performance. In our WAN experiments, AP shows 3x higher peak throughput in both the CFT and BFT comparison, respectively.

Chapter 5

GeoChain: A Locality-Based Sharding Protocol for Permissioned Blockchains

This work proposes a locality-based full sharding protocol, GeoChain, designed for permissioned blockchains. Incorporating geographical property in blockchain sharding not only improves the performance of every single shard but also reduces the ratio of cross-shard transactions in some applications. Both of these improve the scalability of the system. In addition, we propose a client-driven cross-shard transaction processing protocol which is coordinated by the client. We conduct experiments on geo-distributed Amazon cloud to demonstrate the performance.

5.1 Preliminaries

5.1.1 Sharding

Sharding is a technique that tries to distribute storage, computation, and communication to improve scalability. A shard is a ledger that records and processes a subset of all the transactions. Recent research explores full sharding to improve scalability. How to form the shards, distribute transactions, and process cross-shard transactions, remains at the heart of this approach.

Shard formation means assigning participants to shard committees. A typical approach is to randomly assign participants into shard committees [51, 108, 25]. The security of the system largely depends on the size of the committee and the random function it uses. Also, the shard committees have to be reassigned periodically to ensure safety. This is because participants in permissionless blockchains may join and leave at any time. This complicates the process of shard formation. Another approach forms the shards by taking advantage of network proximity [4, 62, 90]. This inspires the design of our GeoChain shard formation.

Transaction placement is another key factor. In partial sharding, transactions can be distributed to any shard as there is a master ledger that handles the cross-shard transactions. A cross-shard transaction is a transaction that involves multiple operations in different shards. But there is no such master ledger in full sharding. Thus, if transactions are randomly distributed, there is a high probability of cross-shard transactions, which limits the scalability of the system. This has been observed in both Omniledger [51] and RapidChain [108]. They use the transaction hash to decide the transaction placement which is essentially a random process. This results in a high rate of cross-shard transactions. RapidChain [108] reports that, for a 500-node network, if the system splits into 3 shards, and the total number of inputs and outputs in a transaction is 3, then 96.3% transactions are cross-shard transactions, while for a 4000-nodes network, if the system splits into 16 shards, then 99.98% transactions are expected to be the cross-shard transactions. These high rates of cross-shard transactions limit performance. OptChain [80] proposes a smart transaction placement strategy to reduce the amount of cross-shard transactions: it puts well-connected transactions into the same shard. Specifically, it constructs a graph, termed *Transactions as Nodes* (TaN), by abstracting each transaction as a node. A directed edge is added to the graph if a transaction uses another transaction as an input. Then it requires the client to calculate a *Temporal fitness* to decide which shard a transaction should be sent to. The *Temporal fitness* score consists of two parts, Transaction-to-Shard score and Latency-to-Shard score. The former is used to determine the probability of placing a transaction into a shard without causing future cross-shard transactions. The latter is used to estimate the processing delay from a transaction to a shard.

The performance of blockchain sharding is highly related to how cross-shard transactions are being processed. OmniLedger [51] proposes an atomic commit protocol, Atomix (Figure 5.1), to handle cross-shard transactions. Suppose that a cross-shard transaction has two inputs, one in shard 1 and one in shard 2, while its output is in shard 3. This is because OmniLedger decides the distribution of a transaction by its hash. In the first step, the client has to verify and lock both inputs from its input shards. If all input shards respond with proof of acceptance, the client commits the transaction by sending the proofs

and commit messages to the input and output shards concurrently. Otherwise, the client has to abort the transaction and unlock the inputs. This requires the client to actively coordinate the process, which involves acquiring proofs and sending commit or abort messages. In addition, the client has to validate the proof. A Byzantine client may lock the inputs forever. THS [25] and Channels [6] propose similar approaches by replacing the client with a reference committee and a trusted channel, respectively. RapidChain [108] proposes a protocol that splits a cross-shard transaction into multiple transactions but requires an inter-shard communication protocol to transfer the inputs from the input shards to the output shard (Figure 5.2). A transaction spends 2 inputs, I_1 and I_2, one from shard 1 and one from shard 2, while the output O belongs to shard 3. Upon receiving the transaction from the client, shard 3 splits the transaction $tx = \{(I_1, I_2), O\}$ into $tx1 = \{I_1, I_1'\}$, $tx2 = \{I_2, I_2'\}$, and $tx3 = \{(I_1', I_2'), O\}$, where I_1' and I_2' are the output of $tx1$ and $tx2$, respectively, and the inputs of $tx3$. Next, shard 3 sends $tx1$ to shard 1 and $tx2$ to shard 2. If both transactions are processed successfully, shard 3 proceeds to process $tx3$. Otherwise, tx is aborted. The main idea is to transfer the inputs from the input shards to the output shard. SharPer [4] also uses a shard as the coordinator to process the cross-shard transaction. But it relies on an all-to-all multicast approach. THS [25] uses a reference committee to handle the cross-shard transactions, which requires an extra effort to form and maintain the reference committee. The reference committee has to make agreement on the transactions which uses another layer of consensus. This hurts the performance. With the high probability of cross-shard transactions in those protocols, the clients suffer a long latency on average. Moreover, clients cannot control the position of the transaction inputs and outputs. This complicates the processing of cross-shard transactions.

5.1.2 Permissioned Blockchain

In contrast to the permissionless blockchain, participants in permissioned blockchains have to establish their identities and be acknowledged by the system administrators. Permissioned blockchains mostly employ classical fault tolerance consensus protocols in ordering the blocks. Hyperledger Fabric (HF) [5] is one of the most popular permissioned blockchain systems that enable smart contract execution. HF has been widely used in logistics, identity management, and the banking industry. Instead of the *order-execute-commit* pattern, HF introduces *execute-order-validate* design to improve the system performance. This architecture not only enables parallel executions but also uses a predefined policy to validate the transactions. Specifically, transactions are first simulated according to the world state to determine the data to be written in the ledger. This is used to verify the correctness of the transactions, which means endorsing. Each result is an endorsement of the transaction.

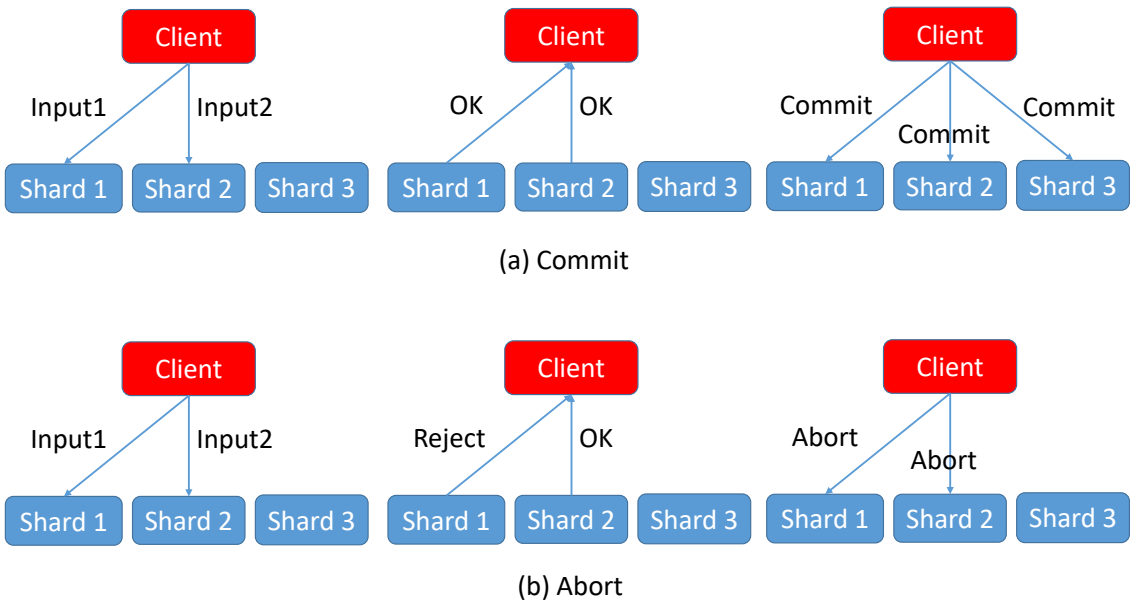


Figure 5.1: Atomix – OmniLedger Cross-Shard Protocol.

Then the transactions are ordered by an independent ordering service. A transaction can be ordered only if it has sufficient endorsements according to an endorsement policy. In the end, the transactions have to be validated and committed in the ledger to resolve race conditions. The whole process is achieved by using different types of nodes that have an assigned role. To achieve this, HF has three different node types, which are peer, endorser, and orderer. Since it is a permissioned network, all nodes have to be acknowledged by a membership service provider (MSP). MSP is a component of HF that governs the identities and associated operations. Otherwise, they will not be recognized by the system. Peers are used for executing and validating transactions. Each honest peer maintains a world state of the ledger, which must be consistent with other peers. A smart contract, which is called a chaincode in HF, is installed on certain peers that are used for executing transactions. The results of the contract are called endorsements. Such peers are also called endorsers. An endorsement policy is used to define which peers have to endorse a transaction. In addition, peers are also responsible for validating the transactions before committing them to the ledger but after ordering. Orderer nodes are used only for ordering transactions. They collectively form an ordering service by running a consensus protocol. It is used to ensure the unique order of the transactions. The orderer nodes collect transactions

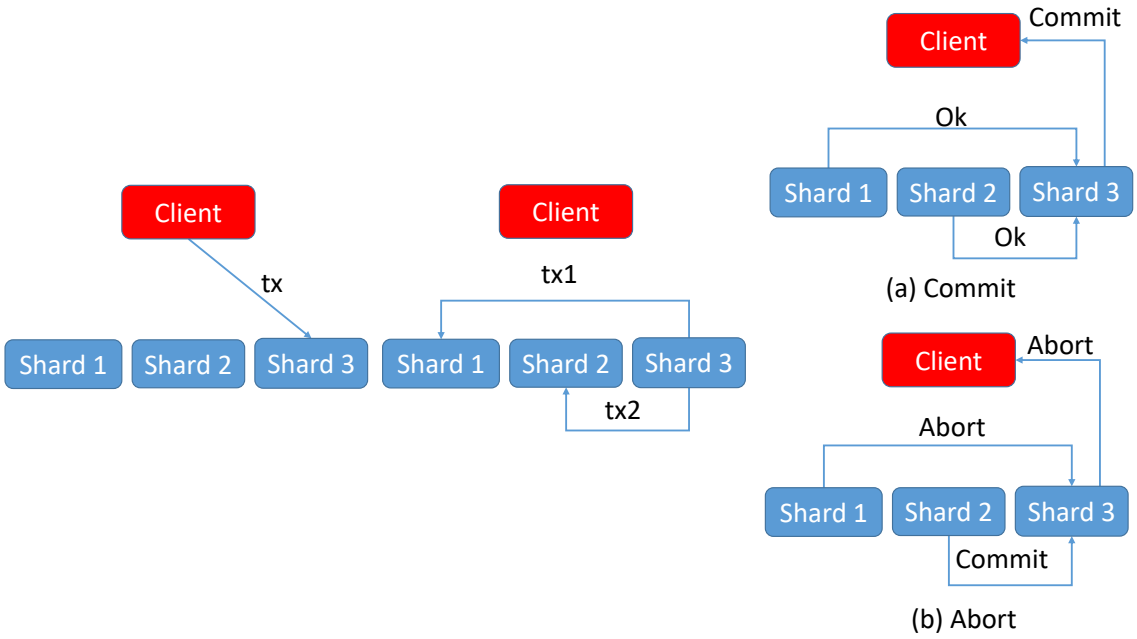


Figure 5.2: RapidChain Cross-Shard Protocol.

associated with endorsements and order them without any knowledge of the ledger state. This allows the ordering service to be independent of peers. Also, the consensus protocol of the ordering service is easy to replace. As a result, HF enables a pluggable consensus mechanism. A transaction is processed at a specific set of peers and orderers that maintain a consistent ledger. Channel [6] is the way of sharding in HF, but all the channels still share the same ordering service. In addition, it requires a trusted channel or Atomix [51] to handle cross-shard transactions.

The transaction workflow follows Figure 5.3. A client proposes a transaction to a sufficient number of endorsers, which are endorsers 1, 2, and 3 in the figure, according to an endorsement policy. The endorsement policy is defined as a logical expression, such as "two out of three". The endorsers simulate the transaction regarding the world state and respond with an endorsement if the simulation succeeds. The client broadcasts the results to the ordering service only after it has collected enough endorsements (at least two out of three). The ordering service is responsible for gathering transactions into blocks and ordering the blocks into a sequence. In the next step, the ordering service delivers the ordered blocks to all peers for final validation. Only valid transactions are committed on the ledger and updated to the world state. The other transactions are aborted.

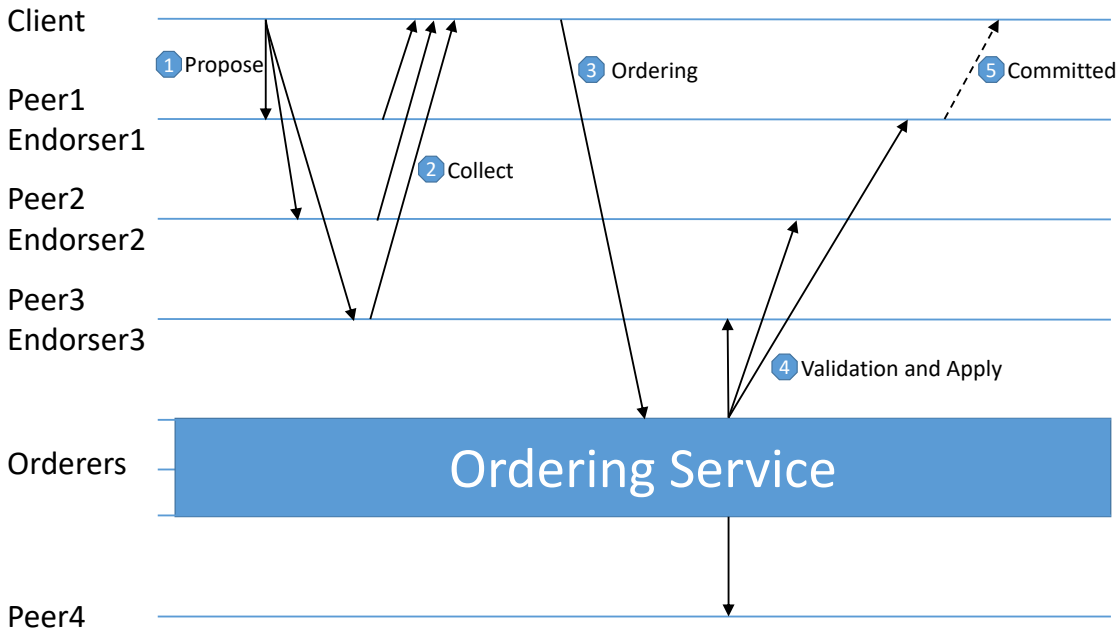


Figure 5.3: Hyperledger Fabric Transaction Flow.

5.1.3 System Model

We rely on the partially synchronous [30] network where there is an unknown upper bound on the message delay. In addition, we assume reliable point-to-point communication channels that cannot lose, corrupt, duplicate, or reorder messages. Furthermore, we are targeting a permissioned network where the identities of the participants, and the servers that maintain the blockchain, have been established already. Overall, there are at least $2f + 1$ participants under the crash failure model (CFT) and $3f + 1$ participants under the Byzantine failure model (BFT), where at most f of them may be faulty. Each client may suffer either crash failure or Byzantine failure. Each client coordinates its own transactions. The transaction follows the UTXO model. UTXO is a typical transaction model in blockchain networks [77, 108, 51]. Each UTXO is uniquely identified by its hash. Each UTXO can only be used once as one of the inputs of another UTXO transaction. Each new block includes multiple UTXO transactions and is recorded in the blockchain. GeoChain ensures safety and liveness.

5.2 Locality Based Sharding Design

This section presents how GeoChain achieves full sharding. The locality information is derived from the established identities of the users.

5.2.1 Shard Formation

For a blockchain without sharding, participants are distributed around the world and work together to build a unified ledger. A major drawback of this approach is that the network latency among the participants may range from less than 1 ms to more than 200 ms. Such 200ms latency drastically degrades the performance of the system. Both Mir-BFT [98] and Red Belly [23] show better performance in the local area network (LAN) than in the wide area network (WAN). In addition, the consensus protocols under a large number of participants suffer weak scalability. The evaluation of MirBFT [98] shows that performance degrades while the number of participants increases in protocols PBFT [19], HoneyBadger [74], and itself. When sharding is introduced, each shard achieves consensus independently with a reduced number of participants which alleviates the scalability bottleneck.

Bearing the above in mind, we incorporate the geographical property, namely locality, in the sharding of permissioned blockchains, which takes advantage of network proximity [4, 90, 62]. The system administrators can perform the configuration and the clustering steps as in other systems [5, 108, 51]. Specifically, each shard is formed based on a regional code that indicates the geographical position of the participants. A regional code is defined as a cluster classifier. Geographically close participants are assigned the same regional code. Example factors to decide the classification are IP address and peer-to-peer network latency. In other words, participants are clustered into shards according to their relative network proximity. This is done by the system administrators when the participants establish their identities. For instance, participants in North America with less than 50ms peer-to-peer network latency are assigned to the same shard. This minimizes the impact of the network latency within each shard which leads to better intra-shard performance. The trade-off is fault tolerance where each shard has to assure intra-shard safety on top of system-wide safety. SharPer [4] introduces this approach and assumes pre-determined fault-tolerant clusters where at most f faulty nodes out of $2f+1$ nodes in the crash failure model and $3f+1$ nodes in the Byzantine failure model. As a result, the fault tolerance depends on the size of the shard committee. Other sharding protocols [51, 108] enforce system-wide fault tolerance and target a high probability of intra-shard safety.

Inputs	A collection of transaction inputs
Outputs	A collection of transaction outputs
Shard	Which shard this transaction belongs to

Table 5.1: Transaction data structure

5.2.2 Transaction Placement

As we stated in Section 5.1, randomized transaction placement [51, 108] leads to a high ratio of cross-shard transactions, which degrades the performance. Considering some practical applications, like banking systems, retail payments, and electric vehicle charging, human activities are mostly restricted by geographic location. Thus, most transactions of such applications are conducted in the same geographical region. McKinsey reports that, on average, merely 11% of global payments revenue came from cross-border transactions in 2020-21 [69]. Bearing this in mind, we propose a locality-based transaction placement strategy that intuitively incurs a low probability of cross-shard transactions. Specifically, we require that each transaction includes shard information, namely the regional code of a shard, which states its placement. Such a transaction can be only processed by the designated shard. Table 5.1 shows the data structure which comprises not only UTXO inputs and outputs but also the shard information the transaction belongs to. The shard information is used to decide which shard handles this transaction. A requirement of such a transaction is that the inputs and the transaction have to belong to the designated shard. Otherwise, the shard cannot verify the transaction. For example, Alice owns a coin with regional code A in one shard, such as shard 1, and transfers this coin to Bob. At the end of the transaction, Bob owns the coin with regional code A in shard 1. However, we may ask what if Bob needs the coin from a shard other than shard 1, which means a transaction from a different shard. Thus, a major challenge of our transaction placement strategy is how to enable and handle cross-shard transactions.

5.3 Cross-shard Transactions

We propose a client-driven efficient protocol to deal with cross-shard transactions. The idea is to give clients flexibility to manipulate their transactions. Each UTXO is assigned the regional code of the shard to which it belongs. We assume that the UTXOs have equal

Algorithm 9: IOCC Client

```
188 Procedure Transfer (input, shard x)
189 |   Propose input to input.shard
190 |   for each non-faulty node P_i in input.shard do
191 |   |   cert_i = Execute_{P_i}(input, x)
192 |   end
193 |   Wait for valid cert from nodes
194 |   return {cert_0, cert_1, cert_2, ...}
195 end
196 Procedure Main (inputs = {input_0, input_1, ...}, outputs, destination shard x)
197 |   for each input_i in inputs do
198 |   |   if input_i.shard is not shard x then
199 |   |   |   proof_i = Transfer(input_i, x)
200 |   |   |   if proof_i is invalid then
201 |   |   |   |   return null
202 |   |   |   end
203 |   |   else
204 |   |   |   proof_i = null
205 |   |   end
206 |   end
207 |   tx = {{(input_0, proof_0), ...}, outputs, x}
208 |   Propose tx to shard x
209 |   for each non-faulty node P_i in shard x do
210 |   |   cert_i=Execute_{P_i}(tx, x)
211 |   end
212 |   Wait for valid cert from nodes
213 |   return {cert_0, cert_1, cert_2, ...}
214 end
```

value across different shards[1]. Notice that, the whole system is still geo-distributed, but each shard is not. We require each client to know where its transactions belong. Thus, if a client wants to propose a transaction to shard x, it has to follow 2 steps as Algorithms 9 and 10.

Step 1. A client first gathers inputs and proposes a transfer transaction for each input in the shard other than shard x. And it requires a *proof* from those shards as line 197 to 206. A transfer transaction is defined as a transaction to transfer a UTXO from one shard to another shard. A *proof* is a quorum certificate from the input shard which can be verified by the destination shard. The quorum certificate is a collection of transaction commitments from the participants of the input shard such as line 194. A proof has to have certificates from at least a majority of shard participants. A majority is $f + 1$ out of $2f + 1$ under the crash failure model and $2f + 1$ out of $3f + 1$ under the Byzantine failure model. The proof is used to establish that a UTXO has been transferred from the input shard to the destination shard. Once a valid *proof* is generated, the UTXO in the input shard is marked as transferred and cannot be rolled back.

Step 2. The client appends the proofs to the original inputs and forms a new transaction such as line 207. Then, the transaction is proposed to and processed by the destination shard. By using this approach, a transaction input in the destination shard must be either a valid UTXO in the same shard or valid proof of UTXO transferred from the input shard to the destination shard as shown in Algorithm 10.

Algorithm 10: IOCC Node in Shard y

```
215 Procedure Execute (tx, x)
216     if tx.shard is not shard y then
217         return null
218     end
219     if tx is valid then
220         Commit and mark that tx has been transferred to shard x
221         return Commitment of tx
222     else
223         return null
224     end
225 end
```

[1]It is possible that the UTXOs in different shards have different values. Then, the system requires an extra mechanism to decide the exchange policy among the shards which is out of the scope of this paper.

Steps 1 and 2 are two related transactions that must be performed in sequence but not necessarily consecutively. The client is only used for assembling and proposing the transactions. As long as the input shards commit the transfer transactions, the proofs are available at any later time. Figure 5.4 shows a concrete example of the above process. A client, who has UTXOs in shard 1 and shard 2, wants to make a transaction to shard 3. The client first sends $tx1$ and $tx2$ to shard 1 and shard 2, respectively. $tx1$ and $tx2$ are used to transfer the UTXOs from shard 1 and shard 2 to shard 3. If both shard 1 and shard 2 commit the transaction and provide *proofs*, $tx1'$ and $tx2'$, which is case 1 in the figure, the client may submit these *proofs* at any later time in shard 3. Once the *proofs* are generated, the UTXOs in shard 1 and shard 2 are treated as transferred implicitly and cannot roll back. After that, the client proposes a new transaction that appends the *proofs* to the original inputs and sends the transaction to shard 3. In case 2, if $tx1'$ aborts, the cross-shard transaction aborts as well. But $tx2$ has been transferred from shard 2 to shard 3 as $tx2'$. The client can hold the proof of $tx2'$ and decide at any later time when to deposit $tx2'$ to shard 3. In another case, if the client crashes in the middle of the protocol, Atomix [51] may lock the inputs forever. However, in our protocol, even if the client crashes before shard 3 receives $tx1'$ and $tx2'$, the proofs are still valid and available in shard 1 and shard 2. When the client recovers, it may extract the *proofs* again and proceed as usual. At this point, shard 3 verifies the *proofs* without communicating to any other shards and confirms the validity of the transaction. As the protocol is essentially moving a UTXO from input shards to the output shard, the protocol is termed as inputs to output cross-shard commit (IOCC).

Double-spending is invalid in IOCC. Within a shard, a block only includes non-conflict transactions, and blocks are chained in a unique sequence. A pair of conflict transactions means that the transactions include at least one identical UTXO in inputs, or an output of a transaction is an input of the other transaction. For cross-shard transactions, a UTXO is either spent in the same shard or transferred to another specific shard. If the UTXO is transferred, the original shard has to provide a *proof* as a commitment that the UTXO has been transferred. The *proof* can only be deposited once in the destination shard. As a result, IOCC ensures no double-spending.

To further optimize the performance, if a client wants to make multiple transfers that use the inputs from the same shard, the transfer transactions can be put into a batched transaction. For example, this happens when a client moves or travels from one country to some other countries (the frequency may vary from daily to yearly). The client batches multiple cross-shard transactions from the input shard to the output shard, and puts these transactions into a single batched cross-shard transaction. This batched transaction is sent to the input shard as a single transaction. The input shard verifies the batched inputs

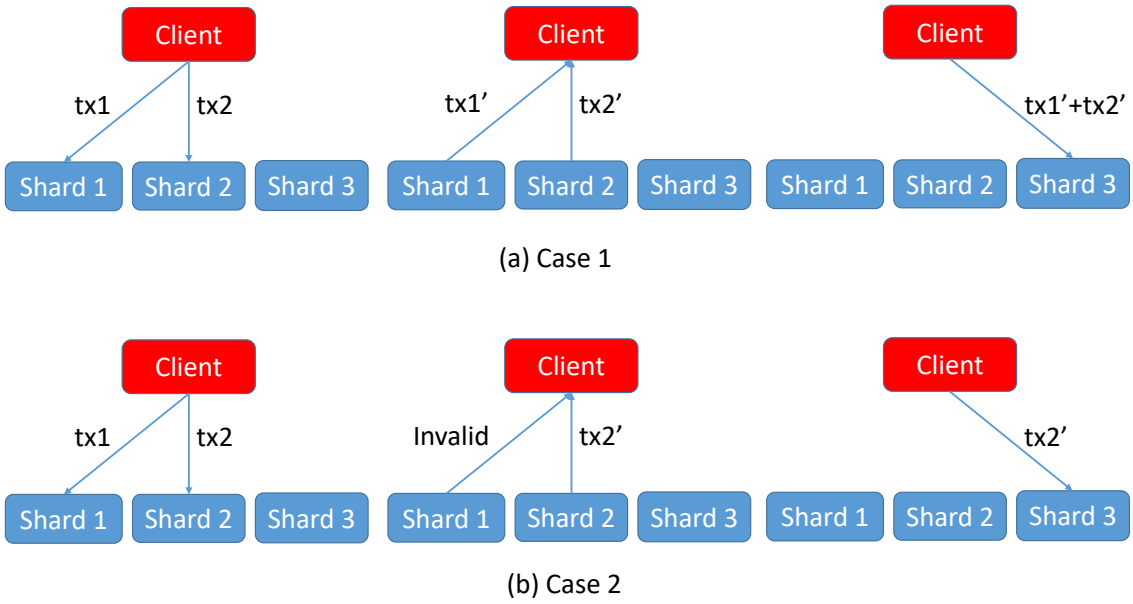


Figure 5.4: IOCC – GeoChain Cross-Shard Protocol.

and returns a batched *proof* to be used in the output shard. As a result, multiple cross-shard transactions are converted to a single transfer transaction and multiple intra-shard transactions, which leads to higher throughput.

Note that IOCC is designed for the UTXO transaction model. However, we can adopt IOCC to the account model by recording one account in multiple shards. This means that each account holder owns multiple accounts located in different shards. Thus, a cross-shard transaction can transfer values among these accounts. Aborting a cross-shard transaction does not roll back the account operations but transfers the value to the account in the destination shard. The trade-off of this is that multiple accounts may be created for the same owner in different shards.

In summary, IOCC splits a cross-shard transaction into multiple intra-shard transactions that are managed by the client directly. First of all, there is no direct inter-shard communication, which reduces the protocol complexity. Also, the client is able to batch cross-shard transactions and assist in achieving high performance. This gives the clients flexibility in assets management which is not possible in RapidChain [25]. Moreover, the double-spending problem is avoided as each UTXO can either be used once in an intra-shard transaction or transferred to another shard. The IOCC protocol also guarantees that

each *proof* can only be successfully deposited once.

5.3.1 Safety and Liveness

GeoChain forms the shard committee using geographical property. Due to the nature of the permissioned network, the identities of the participants are known to the network. We assume that there are at least $2f + 1$ participants under the crash failure model (CFT) and $3f + 1$ participants under the Byzantine failure model (BFT) in each shard, where at most f of them may be faulty. This means that each shard must have at least a certain number of participants to achieve consensus. The fault tolerance depends on the size of the shard committee. Also, different shards may have different numbers of participants. In addition, the client cannot crash forever (e.g. restarting after a crash) and is responsible for completing its own transactions. This means a client is able to deliver the messages eventually. We ensure that:

Theorem 11. *Safety: No invalid transaction is ever decided if each shard achieves shard safety.*

Proof. There exists a consensus protocol that always validates and makes agreement on the order of proposals. Each shard runs such a protocol to process transactions and commits transactions with a consistent order where each transaction is proposed by a client and validated by the non-faulty shard participants. As a result, each shard achieves shard safety.

Suppose each shard achieves the above shard safety, then each shard merely processes two types of transactions: intra-shard transaction and transfer transaction. Since intra-shard transactions do not require any participation of other shards, intra-shard transactions are always safe because of the shard safety. There are two cases of a cross-shard transaction,

Case 1: A transfer transaction moves the UTXO out of the shard. In this case, the shard committee responds with a *proof* if the transaction is valid. Otherwise, the shard committee aborts the transfer transaction. As long as the transaction is committed, the *proof* is always valid due to the shard safety. Even if the client crashes or suffers Byzantine failure after proposing the transaction, the *proof* is still generated and can be extracted after the client is restored.

Case 2: A transfer transaction moves a UTXO in from another shard. Then, the transaction has to include a *proof* from another shard which can be validated. The *proof* defines the source shard and the destination shard. The transaction succeeds if the *proof* is valid, aborts otherwise.

Thus, cross-shard transactions are always safe as well. As a result, safety is ensured. □

Theorem 12. *Termination: Every transaction is either committed or aborted eventually if a client is able to deliver the messages eventually.*

Proof. Suppose there are fewer than 1/2 faulty participants under the crash failure model and 1/3 faulty participants under the Byzantine failure model of a shard committee and there is an unknown upper bound on the message delay, then there exists a consensus protocol [81, 19] that terminates eventually. If every shard runs such a protocol, both the intra-shard transaction and the transfer transaction are committed or aborted eventually. In addition, if a client is able to deliver the messages eventually, a cross-shard transaction is committed or aborted eventually. □

Note that, according to the analysis in the previous section, even if a Byzantine or crashed forever client may partially complete a cross-shard transaction, safety is still assured.

5.4 ACID properties

In the context of a traditional database system, a transaction is a collection of read/write operations that access and possibly modify the data in the system. ACID [13] describes four required properties of transactions, which are atomicity, consistency, isolation, and durability.

Atomicity: all the operations within a transaction are either performed completely or not at all.

Consistency: the state of the system must be consistent and valid before and after the transaction.

Isolation: the result of processing multiple transactions concurrently is the same as if the system processes the transactions one by one in some order. The widely adopted technical definition of this property is serializability [82].

Durability: once a transaction completes, the changes to the system persist and cannot be revoked, even in the case of system failure.

In a database system, ACID properties refer to the transactions in the logical layer where the logical schema defines the logical relationships and constraints over the tables. For example, a transaction of transferring $100 from Alice to Bob involves two operations,

debit $100 from Alice's account and credit $100 to Bob's account. After processing the transaction, there are two possible outcomes. If the transaction is fully committed, then all the changes to the state of the database, Alice pays $100 to Bob, have been recorded. If a transaction is aborted, the state of the database, the account balance of Alice and Bob, does not change. This means that the system either moves to a new state or remains in the old state, which ensures atomicity. Note that the state above refers to the logical state which describes the logical relationships of the transactions. In contrast, the physical state, the actual data stored, may change even if the transaction is aborted since ACID properties do not restrict how the database physically records transactions. Normally, a logical schema can be implemented with different physical schema in traditional database systems.

A blockchain is an append-only ledger system where a new block contains multiple transactions appended to the end of the ledger. A transaction within a block is a collection of data that describes one or more application-specific operations, such as transferring ownership, recording originality, and authorizing accessibility. A blockchain enforces a specific physical schema, the block data structure, to implement different logical schema.

Considering the ACID properties in a non-sharded blockchain system, the atomicity is guaranteed as each valid transaction is either fully committed and appended to the system, or not at all. In addition, since a blockchain system appends blocks one at a time, and all the transactions within a block are executed sequentially, then the transactions are serializable, which ensures isolation. Consistency is ensured because valid transactions are appended in a unique order. For durability, consensus protocols are used to ensure that an appended block is persisted in a tamper-resistant manner.

When considering a sharded blockchain system, there is no difference in durability, as each shard chain follows the rule of only appending a sequence of blocks to its shard. However, atomicity, isolation and consistency are not that straightforward because the system maintains multiple chains of blocks, where transaction execution is no longer serial. Specifically, a cross-shard transaction involves multiple intra-shard transactions in different shards. The transactions in one shard may rely on some other transactions in other shards. Thus, we must also consider the cross-shard transactions when discussing atomicity and isolation. Consistency is guaranteed if atomicity and serializable isolation are achieved. This is because the committed transactions are valid and serializable.

Let's first introduce the UTXO transaction model in the GeoChain protocol. The logical schema is that each client owns some UTXO assets. There are three states for each UTXO, which are *Non-existent*, *Unspent*, and *Spent*. Each intra-shard transaction logically includes a pair of read and write operations for each input and one write operation for its output.

These operations are used to update the physical state of the committed transactions. If two operations access the same UTXO, and at least one of the operations is write, then these two operations are called conflicting operations. A cross-shard transaction consists of multiple intra-shard transactions in different shards. Initially, the system issues some genesis UTXOs. Then, an input of a transaction is either another UTXO in the same shard or a *proof* of another UTXO in a different shard. The output of the transaction is itself. For example, Table 5.2 shows an initial physical state of the system where Alice owns an unspent *UTXOA* in shard 1. If Alice wants to transfer *UTXOA* to Bob in shard 2, the cross-shard transaction is abstracted as two intra-shard transactions that execute in sequence, which are [read(*UTXOA*), write(*UTXOA*), write(*UTXOB*)] in shard 1, and [read(*UTXOB*), write(*UTXOB*), write(*UTXOC*)] in shard 2. In the first transaction, if *UTXOA* exists and is *Unspent* in shard 1, then the state of *UTXOA* is updated as *Spent*. Next, *UTXOB* is inserted as the proof that *UTXOA* has been transferred from shard 1 to shard 2. We mark the state of *UTXOB* in shard 1 as *Spent* to indicate that *UTXOB* cannot be spent in shard 1 anymore. Then, shard 2 has to first verify that *UTXOB* does not exist. This is to ensure that *UTXOB* can only be deposited once in shard 2. No other shards can deposit *UTXOB* as it is designed as a transfer from shard 1 to shard 2. After that, *UTXOB* is verified and marked as spent in shard 2. As a consequence, *UTXOC* is created and given to Bob, such as Table 5.3. To avoid double-spending, we mark the state of *UTXOB* as *Spent* in shard 2. Note that *UTXOB* is the output of the transaction in shard 1 but the input of the transaction in shard 2.

Asset	State	Owner	Shard
UTXOA	Unspent	Alice	1

Table 5.2: Initial physical state

Asset	State	Owner	Shard
UTXOA	Spent	Alice	1
UTXOB	Spent	Alice	1
UTXOB	Spent	Alice	2
UTXOC	Unspent	Bob	2

Table 5.3: Physical state after a transaction

In sharded blockchains, transactions may be placed in multiple shards. When processing a cross-shard transaction, if the transaction is committed, the system moves to a new logical

state as normal. But if the transaction is aborted, the physical state may still change. The logical state describes the application-specific logical relationship of the transactions. For example, initially, Alice owns a coin C while Bob owns nothing. The logical schema here is how many coins own by Alice and Bob, respectively. Then, Alice wants to transfer the coin C from shard A to Bob in shard B. If this transaction is committed, C is recorded as C' in shard B that belongs to Bob. C and C' are coins in different states, both logical and physical. However, if the transaction is aborted, there may be two outcomes: 1) Nothing changes, C is still in shard A and belongs to Alice. 2) C has been transferred to shard B as C'' but still belongs to Alice. In the first case, nothing changes to either the logical state or the physical state. But, in case 2, C (before the transaction) and C'' (after the transaction) are coins in different physical states but equal logical states. This is because C still belongs to Alice but shard B records it as a new coin, C'', instead of C in shard A. As a result, Alice either owns C as case 1 or owns C'' as case 2. And Bob still has no coins. The logical schema does not change. Then, we say that these two cases both satisfy the atomicity property.

In a non-sharded blockchain, serializability is guaranteed by appending valid transactions in a single chain of blocks. However, in sharded blockchain systems, concurrent transactions happen in different shards. Serializability has to consider not only intra-shard transactions but also cross-shard transactions. According to [94], we want to prove that the schedule of a set of committed transactions is conflict serializable. A schedule is a sequence of read/write operations of the transactions. A schedule is conflict serializable iff its precedence graph is acyclic. The precedence graph is derived from the schedule. It contains a node for each committed transaction and a directed edge for each pair of conflicting operations in the schedule.

According to the algorithm described in Section 5.3, IOCC ensures that:

Theorem 13. *Atomicity: A transaction is either fully committed or not at all.*

Proof. Intra-shard transactions are either committed or aborted as a whole. Thus the atomicity is ensured.

A cross-shard transaction, which involves multiple shards, is split into multiple transfer transactions in the input shards and one intra-shard transaction in the output shard. A cross-shard transaction fully commits only if all mentioned transactions commit. If any transfer transaction aborts, the intra-shard transaction aborts as well as the cross-shard transaction. However, in this situation, some transfer transactions may still commit. The UTXOs of these transactions are transferred to the output shard without changing the logical schema. In another case, if only the intra-shard transaction aborts, the cross-shard

transaction aborts. However, the transfer transactions commit and the UTXOs have been transferred to the output shard without changing the logical schema. Thus, the cross-shard transaction aborts as if the transaction was not processed.

Thus, the atomicity is ensured. □

For example, in Figure 5.4, if either $tx1$ or $tx2$ fails, like in case 2, the cross-shard transaction aborts. But $tx2'$ or $tx1'$ can still be used in any other transaction that belongs to shard 3. The atomicity is ensured as the client owns either $tx1'$ or $tx2'$ as a replacement of $tx1$ or $tx2$. If both $tx1$ and $tx2$ fail, nothing happens to any shard.

Theorem 14. *Serializability: For any set of committed transactions, the schedule of these transactions is conflict serializable.*

Proof. According to [94], we want to prove that the precedence graph of the schedule of these committed transactions is acyclic. A precedence graph contains a node for each committed transaction in the schedule, and a directed edge is added for each pair of conflicting operations. Here we prove that this graph is acyclic, which implies serializability.

Suppose a precedence graph G derived from a schedule of committed transactions $T_1, T_2, ..., T_n$ has a cycle, and the cycle is
$T_1 \to ... \to T_k \to ... \to T_1$. For any pair of transactions, $T_i \to T_j$, in the cycle, where $\to$ means that an operation of T_i precedes and conflicts with an operation of T_j, there are four cases as follows:

1. T_i and T_j conflict on the same transaction input. Then the double-spending problem occurs, which is impossible by design.

2. T_i and T_j conflict on the same transaction output. This means that T_i and T_j have the same output and are the same transaction. This is a contradiction since T_i and T_j are distinct.

3. T_i and T_j conflict on X where X is an input of T_i and the output of T_j. This means T_j finishes before T_i reads its input. This contradicts $T_i \to T_j$.

4. T_i and T_j conflict on Y where Y is the output of T_i and an input of T_j. This means T_i finishes before T_j reads its input, and before T_j finishes. Since this is the only case that does not lead to a contradiction, $T_i \to T_j$, in general, implies that T_i finishes before T_j finishes, and hence it is a transitive relation of transactions in the cycle. Thus it follows from $T_1 \to ... \to T_k \to ... \to T_1$, that $T_1 \to T_1$, meaning that T_1 finishes before itself, which is a contradiction.

Therefore, the precedence graph is acyclic, and conflict serializability is achieved. □

Considering the example in Figure 5.4, if another transaction $tx2''$ wants to transfer the same UTXOs as $tx2$ in shard 2 concurrently, and if both $tx2$ and $tx2''$ are valid, then either $tx2$ or $tx2''$ can be successfully appended. If both $tx1$ and $tx2$ succeed, then the workflow follows case 1 in the figure but $tx2''$ fails. However, if $tx1$ fails and $tx2$ succeeds before $tx2''$, the workflow follows case 2 and $tx2''$ fails. This is the case where both conflict transactions fail.

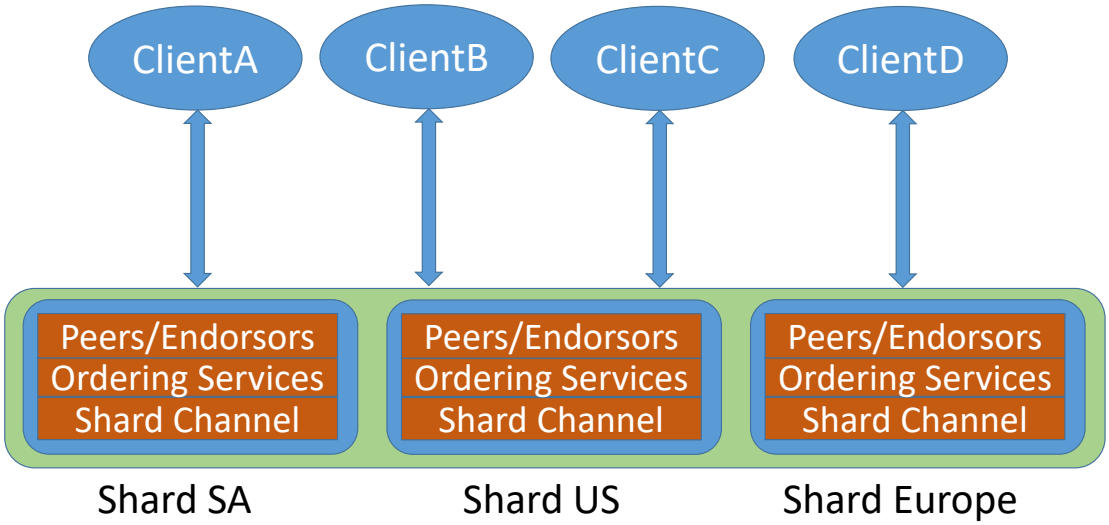


Figure 5.5: Example of 3 Shards GeoChain.

5.5 Evaluation

To investigate the feasibility and evaluate performance, a GeoChain prototype is developed based on Hperledger Fabric v2.3 [5], a popular permissioned blockchain platform. According to the design, a node is assigned into a shard according to its geographical property. We assign the nodes from the same datacenter into the same shard. Each shard comprises a non-overlapping subset of peer nodes and orderer nodes. In addition, each shard runs an independent ordering service that employs the RAFT [81] consensus protocol. The shard ordering service consists of ordering nodes within each shard. RAFT is the consensus protocol in HF v2.3 for ordering transactions. BFT consensus protocol is under development

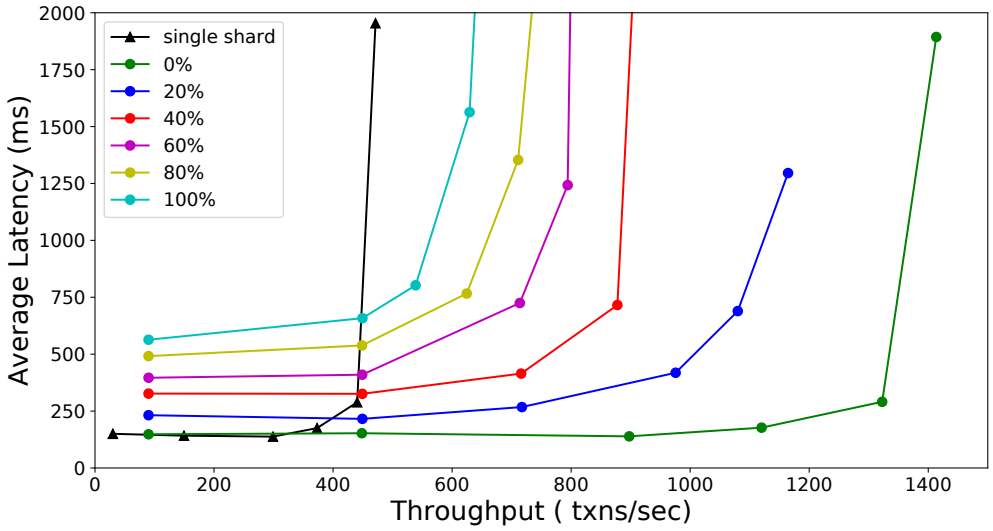


Figure 5.6: Performance under different ratios of cross-shard transactions.

and not mature enough [98]. Furthermore, each shard maintains a unique channel that builds a shard ledger. The channel-specific endorsement policy is defined as *Majority* which means that at least $(n+1)/2$ endorsers have to correctly endorse a transaction. Each ledger merely processes transactions that belong to its shard. A client has to figure out which shard it has to communicate. Otherwise, the transaction is not valid. Figure 5.5 gives an example of this architecture with 3 shards, which are shard North America (NA), shard South America (SA), and shard Europe. Since the proposal response of HF contains the transaction commitments from different peers which satisfies the endorsement policy, it is being adopted as the *proof* for the cross-shard transaction, as we described in Section 5.2. This proof can be verified by any shard, but can only be applied in a specific shard to claim the UTXO transfer. We develop a chaincode that mimics a simplified UTXO transaction. An intra-shard transaction transfers UTXOs from one to another in the same shard, while a cross-shard transaction transfers UTXOs from one to another in different shards.

We evaluate our protocol on Amazon EC2 using c4.xlarge instances (7.5GB memory, 4 cores, 2.9GHz Intel Xeon E5-2666 v3 CPU). The operating system in each instance is Amazon Linux 2. We use Golang version 1.14.15, Docker version 19.03.12, docker-compose version 1.28.6, and HF version 2.3.1. Peers and orderers are running in different instances and deployed using Docker swarm mode. Docker is a platform for developing and running applications. Each application is packaged in an isolated environment, called a container.

Containers contain everything the application needs to run. Docker swarm is a cluster management module to coordinate different docker hosts. Docker swarm simplifies the deployment of containers in a cluster of machines. We create multiple clients to send transactions in an open loop concurrently. Transactions are uniformly distributed to each shard. The throughput is indirectly controlled by adjusting the transaction sending rate. We form each transaction as an ownership transfer of a UTXO asset. The clients control whether the input and the output of the transaction belong to the same shard. This mimics the fundamental transfer of UTXO transaction flow. A cross-shard transaction means that the input and the output belong to different shards. By using this approach, we vary the ratio of cross-shard transactions. The scalability is examined by measuring the throughput and latency with different setups. Table 5.4 lists the configuration parameters of each shard in our GeoChain system. We use *majority* endorsement policy, which is 2 out of 3 organizations in each shard.

Peers per Shard	3
Orderers per Shard	3
Batching Timeout	100ms
MaxBatchCount	100
MaxBatchSize	10MB
Endorsement Policy	MAJORITY
TLS Communication	Enabled
Database	LevelDB

Table 5.4: Configuration

In the first experiment, we deploy the network across multiple AWS EC2 datacenters located on different continents: United States East (USE), South America East (SAE), and Central Europe (EUC). The round-trip average network latency is quantified in Table 5.5. We conduct a set of 7 experiments and summarize the results in Figure 5.6. As a baseline, we first measure the throughput and latency of a single-shard setup without cross-shard transactions. The single shard system is deployed in USE datacenter only. The peak throughput is less than 500 txns/sec [12] while the average latency is around 100ms which is mainly dominated by the batching timeout. This is because all nodes are located in the same region where the network latency is minimized. This demonstrates the fundamental performance of HF. Bearing this in mind, we run a network with 3 shards. Each shard is distributed in a single datacenter which has the same locality. We measure

the performance by varying the ratio, from 0% to 100%, of cross-shard transactions. From figure 5.6, we can see that the peak throughput decreases as we increase the ratio of cross-shard transactions from 0% to 100%. The peak throughput at 0% is about 1400 txns/sec, which approximates triple the throughput of the single-shard network. When all the transactions are cross-shard transactions, the peak throughput is about 900 txns/sec which is still approximately double the baseline throughput. The root reason for this is that each cross-shard transaction actually involves two steps according to our design, which are one transfer transaction in the input shard and one intra-shard transaction using the proof in the output shard. Furthermore, the latency increases from about 100ms to 500ms as we increase the ratio. This reflects the long waiting time for each cross-shard transaction, which not only involves the transaction processing time but also the cross datacenters wide area network (WAN) latency.

	USE	SAE	EUC
USE	<1ms, 870Mbps		
SAE	123ms, 860Mbps	<1ms, 870Mbps	
EUC	99ms, 860Mbps	200ms, 830Mbps	<1ms, 870Mbps

Table 5.5: Round-trip average network latency and bandwidth

Next, we compare the performance of the cross-shard transaction processing protocols between IOCC, batched IOCC, and Atomix. In batched IOCC, we batched 10 cross-shard transactions. We implement Atomix in HF's chaincode as shown in Figure 5.1. It is derived from Appendix D of Omniledger [51]. The *proof* is implemented using transaction commitments of HF for committing or aborting the transaction. Other protocols either use an extra layer of consensus [25, 6] or rely on inter-shard communication protocol [108, 4] which makes the process more complex. Figure 5.7 plots the throughput and latency between these three protocols under 89% and 11% cross-shard transactions, respectively. 89% is the ratio of the cross-shard transaction if the transaction is distributed using the random approach according to RapidChain [108]. 11% is the average ratio of cross-border payments according to the 2021 McKinsey global payments report [69]. Both results show that IOCC outperforms Atomix with higher peak throughput. The reason is that IOCC avoids the step "unlock to commit" as we explain in Section 5.2. IOCC achieves around 50% higher peak throughput at 89% cross-shard rate and around 20% higher at 11% cross-shard rate than Atomix. Batched IOCC achieves even higher peak throughput and lower latency. Specifically, batched IOCC achieves more than 2 times peak throughput compared to Atomix at 89% cross-shard rate and about 30% at 11% cross rate. In addition, comparing the performance between 89% and 11%, the locality-based transaction placement approach shows both higher throughput and lower latency than the randomized approach.

In summary, the experiments show the performance of the GeoChain protocol. However, the ratio of cross-shard transactions highly depends on the applications. In applications like retail payment and electronic vehicle charging, where the ratio should be low, GeoChain is able to achieve near-linear scalability.

5.6 Related Works

Sharding is a promising approach to solving the scalability problem of blockchain systems. We have reviewed some works [51, 108, 25, 6, 4, 80] in Section 5.1 on how they form the shards, distribute and process the cross-shard transactions. There are other sharding protocols for various blockchain applications.

RSCoin [70] is a sharding-based central bank cryptocurrency. The central bank is a trusted entity that authorizes shard committees and audits transactions. Each shard builds a low-level chain of blocks. After a pre-defined time interval, the lower-level blocks are merged into a higher-level chain of blocks by the central bank. RSCoin relies on a trusted party, the central bank, which is essentially not a decentralized system. In addition, RSCoin uses a variation of the two-phase commit protocol to achieve consensus in each shard, which is not Byzantine fault-tolerant and may result in a double-spending.

Elastico [61] is a secure sharding protocol for permissionless blockchains that divides all the processors into several disjoint committees. Each committee processes a disjoint set of transactions and runs in epochs. Each epoch makes agreement on a set of transactions. A global random string is generated to be used in the process of PoW to determine the committees. Then, each committee runs a consensus protocol, PBFT [19], to agree on the set of transactions. At last, each committee broadcasts its transaction set to all the members to achieve final consensus and record the transactions. This is achieved by simply taking the set union of all valid transactions. However, the committee size is relatively small which is due to the overhead of PBFT [19], and increasing the committee size also increases the failure probability. In addition, the committees have to be rebuilt in each epoch, which incurs a latency overhead. Also, using a master ledger to interleave transactions limits performance in the WAN [68].

Chainspace [2] is a sharded smart contracts platform. Smart contracts are used to assign nodes to shards. Each shard runs a BFT protocol, BFT-SMART [14], and maintains a subset of transactions. It introduces a complex mechanism to handle cross-shard transactions where the shards have to coordinate the transaction processing. As a result, Chainspace is not scalable as it can only process around 300 transactions per second in

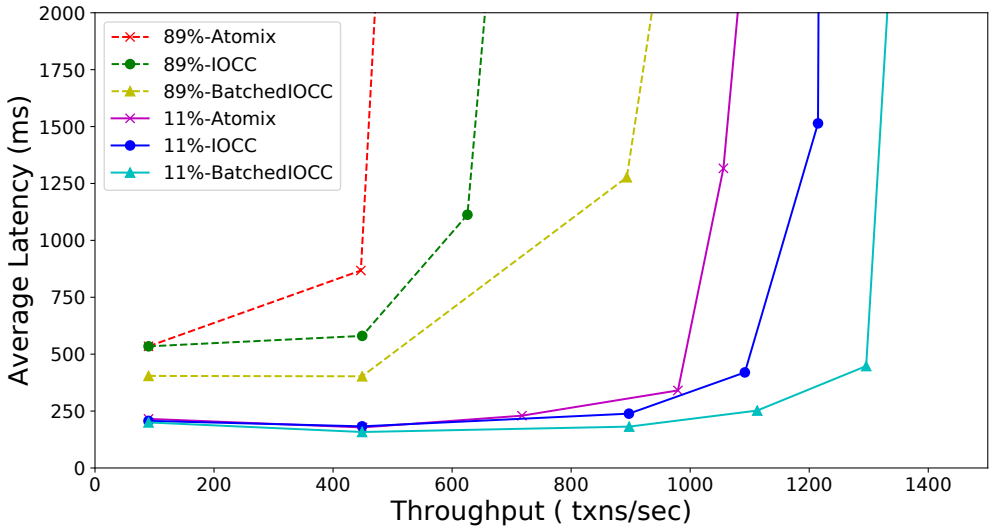


Figure 5.7: Comparison of cross-shard protocol between IOCC and Atomix

a 10-shard setup in its evaluation. Also, although the system ensures safety in all cases, there is a high rate of aborted transactions under high contention.

RedBelly [23] uses DBFT [21] and shard verification to build a blockchain system. DBFT is an efficient Byzantine consensus protocol using a weak coordinator to obtain excellent performance. To optimize the performance, it achieves set Byzantine consensus by running multiple binary consensus instances concurrently with multiple leaders and reconciling to a superblock. Another feature of RedBelly is the sharded verification. Instead of verifying all the transactions in every participant, RedBelly only requires a subset of participants to perform the verification which optimizes the usage of computational resources. Red Belly achieves security, fairness, and scalability. However, a slow leader may dominate the performance of the set Byzantine consensus. Also, the security is conditioned on the number of required verifiers. The experiments show that the performance is limited by the slowest node, and a higher number of verifiers reduces the throughput by half compared to a lower number of verifiers in its single datacenter experiment.

5.7 Conclusion

Sharding is a well-known solution to the scalability problem of blockchain systems. In this paper, we clarify the limitations of present sharding techniques in blockchain systems. Then, we propose a locality-based sharding protocol, GeoChain, to improve the scalability of permissioned blockchains. We use the geographical property to form the shard and decide the transaction placement. Also, we introduce a simple and efficient cross-shard transaction processing mechanism. We implement a prototype based on Hyperledger Fabric and evaluate the protocol on Amazon EC2. The experiments demonstrate the performance of GeoChain through different ratios of cross-shard transactions. GeoChain is a promising architecture in applications like banking and retail payments.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

Blockchain become a promising technology in the coming decades due to the success of Bitcoin. However, scalability is one of the main challenges that limit performance. Sharding technique is being investigated to improve blockchain scalability. This dissertation is focusing on the sharding techniques in permissioned blockchains which includes three pieces of research. Specifically, we make the following contributions:

The first part is the techniques of shard interleaving in partial sharding. We compare the performance and fault tolerance trade-offs when interleaving shards. Although strong coupling is prone to the shard growth rate, it still outperforms weak coupling in a geo-distributed environment. This is because the high latency of the network limits the performance of the consensus layer in weak coupling. In contrast, strong coupling interleaves the shards directly in a round-robin manner. We implement a prototype in Golang and conduct a set of experiments on Amazon to demonstrate our analysis. The result holds even after we augment strong coupling with convergence module, a slow-path mechanism to deal with shard failures. Thus, strong coupling is superior to weak coupling in geo-distributed environments with an even shard growth rate. Strong coupling with convergence module helps native design handle failures.

To further explore the convergence module, we propose Antipaxos to achieve consensus. We first introduce and define k-Interactive Consistency as a new agreement problem. Then, we present the algorithms which reach agreement in a fast path, two rounds of all-to-all communication under the failure-free case and consult decision module to handle failures.

We prove the safety and liveness properties. We implement prototypes of the algorithms and conduct experiments on the Amazon cloud to demonstrate the performance. The results show that if the system runs on the fast path most of the time, Antipaxos achieves high performance compared to state-of-the-art algorithms.

Due to the drawbacks of partial sharding, recent research is targeting full sharding which tries to distribute computation, communication, and storage. In the third part, we propose a locality-based full sharding protocol, GeoChain, in a permissioned blockchain. We leverage the network proximity to cluster participants in multiple shards. The network latency in each shard is optimized by using the locality of the participants. In addition, to reduce the probability of cross-shard transactions, we propose to include the shard information in the transaction data. This helps in transaction placement. For some applications, this leads to a low ratio of cross-shard transactions. To optimize the performance, we propose a client-driven cross-shard transaction protocol, IOCC. This protocol is essentially transferring UTXO from the input shard to the output shard before making any transaction in the output shard. We implement our GeoChain protocol on top of a popular permissioned blockchain network, Hyperledger Fabric, and conduct experiments to demonstrate the performance. The results show that GeoChain achieves high scalability for applications such as asset transfer and retail payments.

6.2 Future Works

The research in this thesis motivates some directions for future works as follows:

1. To further compare STC and WTC, it makes sense to experiment with skewed workloads where some shards grow faster than others.

2. Although AP is capable of handling failures, it has to seek assistance from the DM. We propose and examine different strategies on how to handle failures. However, we can still detect a performance gap when a failure occurs. Thus, to further optimize the performance during a process failure, or network partition, we need to examine and optimize how we process the decisions from the DM and how the DM handle failures.

3. GeoChain achieves high performance because of the assistance of the clients in transaction management and coordination. Although it makes sense to let clients manage the transactions belonging to themselves, a better approach is to find an incentive mechanism for the clients to participate and balance the workload. In addition, we assume that the UTXOs in different shards have the same value which simplifies the cross-shard transactions. However, it is not necessary to make such an assumption. Each shard can be built

independently with various UTXO transactions. At this point, we need the policy to define how to transfer among different shards. Moreover, load balancing will be a potential practical problem if more and more clients transfer on some popular shards. One approach to solve this problem is to split one shard into multiple shards to balance the workload. The key protocol is how and when to split the shards to lessen the effects on the clients while still maintaining safety and liveness.

References

[1] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected $O(1)$ rounds, expected $O(n^2)$ communication, and optimal resilience. *Cryptology ePrint Archive, Report 2018/1028*, 2018.

[2] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *Network and Distributed Systems Security (NDSS) Symposium*, 2018.

[3] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. *IEEE Transactions on Dependable and Secure Computing (DSN)*, 7(1):80–93, 2008.

[4] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In *International Conference on Management of Data*, pages 76–88, 2021.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, and et al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *13th European Conference on Computer Systems Conference (EuroSys)*, pages 1–15, 2018.

[6] Elli Androulaki, Christian Cachin, Angelo De Caro, and Eleftherios Kokoris-Kogias. Channels: Horizontal scaling and confidentiality on permissioned blockchains. In *European Symposium on Research in Computer Security*, pages 111–131, 2018.

[7] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timon, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. *URL: http://www. opensciencereview.*

com/papers/123/enablingblockchain-innovations-with-pegged-sidechains, 72:201–224, 2014.

[8] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. SoK: Consensus in the Age of Blockchains. In *1st ACM Conference on Advances in Financial Technologies (AFT)*, pages 183–198, 2019.

[9] Catalonia-Spain Barcelona. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 369–384, 2008.

[10] Michael Ben-Or. Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In *2nd annual ACM symposium on Principles of distributed computing (PODC)*, pages 27–30, 1983.

[11] Michael Ben-Or and Ran El-Yaniv. Resilient-Optimal Interactive Consistency in Constant Time. *Distributed Computing*, 16(4):249–262, 2003.

[12] Caliper Benchmarks. Hyperledger blockchain performance. *https://hyperledger.github.io/caliper-benchmarks/*.

[13] Philip A Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 2009.

[14] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State Machine Replication for the Masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 355–362, 2014.

[15] Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[16] Eric Brewer. Towards Robust Distributed System. In *Symposium on Principles of Distributed Computing (PODC)*, volume 7, pages 343477–343502, 2000.

[17] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *https://arxiv.org/abs/1807.04938*, 2019.

[18] Vitalik Buterin. Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform. 2014.

[19] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *3rd Symposium on Operating Systems Design and Implementation (OSDI)*, volume 99, pages 173–186, 1999.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. *The MIT Press, 3rd edition*, pages 254–256, 2009.

[21] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains. In *17th IEEE International Symposium on Network Computing and Applications (NCA)*, pages 1–8, 2018.

[22] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating the Red Belly Blockchain. *arXiv:1812.11747 [cs.CR]*, 2018.

[23] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red Belly: A Secure, Fair and Scalable Open Blockchain. In *42nd IEEE Symposium on Security and Privacy*, pages 466–483, 2021.

[24] Michael Crosby, Nachiappan, Pradhan Pattanayak, Sanjeev Verma, and Vignesh Kalyanaraman. Blockchain technology: Beyond bitcoin. *Applied Innovation Review*, pages 6–19, June 2016.

[25] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards Scaling Blockchain Systems via Sharding. In *International Conference on Management of Data (SIGMOD)*, pages 123–140, 2019.

[26] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security, Christ Church, Barbados, Springer*, pages 79–94, 2016.

[27] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Kian-Lee Tan, and Beng Chin Ooi. Blockbench: A benchmarking framework for analyzing private blockchains. In *2017 ACM International Conference on Management of Data*, pages 1085–1100, 2017.

[28] Florian Dold and Christian Grothoff. Byzantine Set-Union Consensus Using Efficient Set Reconciliation. In *Proceedings of International Conference on Availability, Reliability and Security (ARES)*, pages 29–38, 2016.

[29] Assia Doudou and André Schiper. Muteness Detectors for Consensus with Byzantine Processes. In *Proceedings of Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 315, 1998.

[30] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[31] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-Machine Replication for Planet-Scale Systems. In *Proceedings of European Conference on Computer Systems (EuroSys)*, pages 1–15, 2020.

[32] Ethcore. Parity: next generation Ethereum browser. *https://ethcore.io/parity.html.*

[33] Ethereum. Ethereum foundation sharding-faq. *https://eth.wiki/sharding/Sharding-FAQs.*

[34] 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.

[35] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *26th Symposium on Operating Systems Principles (SOSP)*, pages 51–68, 2017.

[36] Seth Gilbert and Nancy Lynch. Brewers Conjunction and the Feasability of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News*, 33:51–59, 2002.

[37] Martin Gosele and Philipp Sandner. Analysis of Blockchain Technology in the Mobility Sector. *FSBC Working Paper, Frankfurt School Blockchain Center*, April 2018.

[38] Rachid Guerraoui, Nikola Knežević, Vivien Marko Quéma, and Vukolić. The Next 700 BFT Protocols. In *Proceedings of European Conference on Computer Systems (EuroSys)*, pages 363–376, 2010.

[39] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, and Alin Tomescu Orr Tamir. SBFT: A Scalable and Decentralized Trust Infrastructure. In *49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580, 2019.

[40] Adiseshu Hari and T.V. Lakshman. The internet blockchain: A distributed, tamper-resistant transaction framework for the internet. In *15th ACM Workshop on Hot Topics in Networks. ACM*, pages 204–210, 2016.

[41] Faiza Hashim, Khaled Shuaib, and Nazar Zaki. Sharding for scalable blockchain networks. *SN Computer Science*, 4(1):1–17, 2023.

[42] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS)*, volume 70, pages 25:1–25:14, 2017.

[43] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of USENIX Annual Technical Conference (ATC)*, page 11, 2010.

[44] Hyperledger. Measuring blockchain performance with hyperledger caliper. *https://www.hyperledger.org/blog/2018/03/19/measuring-blockchain-performance-with-hyperledger-caliper*, 2018.

[45] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 425–438, 2016.

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

[47] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 245–256, 2011.

[48] Srinivasan Keshav, Wojciech Golab, Bernard Wong, Sajjad Rizvi, and Sergey Gorbunov. RCanopus: Making Canopus Resilient to Failures and Byzantine Faults. *arXiv:1810.09300 [cs.DC]*, 2018.

[49] Uri Klarman, Soumya Basu, Aleksandar Kuzmanovic, and Emin Gun Sirer. bloXroute: A Scalable Trustless Blockchain Distribution Network whitepaper v1.0. *Bloxroute Labs, Evanston, IL, USA, White Paper*, Mar 2018.

[50] 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)*, pages 279–296, 2016.

[51] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *IEEE Symposium on Security and Privacy*, pages 19–34, 2018.

[52] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Operating Systems Review (SIGOPS)*, 41:45–58, 2007.

[53] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011.

[54] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE transactions on software engineering*, pages 125–143, 1977.

[55] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News*, pages 51–58, 2001.

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

[57] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *Journal of the ACM (JACM)*, 30(3):668–676, 1983.

[58] Peilun Li, Guosai Wang, Xiaoqi Chen, Fan Long, and Wei Xu. Gosig: A Scalable and High-Performance Byzantine Consensus for Consortium Blockchains. In *Proceedings of ACM Symposium on Cloud Computing (SOCC)*, pages 223–237, 2020.

[59] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing. *IEEE Transactions on Computers*, 68(1):139–151, 2018.

[60] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and Secure Global Payments with Stellar. In *27th ACM Symposium on Operating Systems Principles*, pages 80–96, 2019.

[61] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A Secure Sharding Protocol For Open Blockchains. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 17–30, 2016.

[62] Sidra Malik, Salil S Kanhere, and Raja Jurdak. Productchain: Scalable Blockchain Framework to Support Provenance in Supply Chains. In *IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–10, 2018.

[63] Stathis Maneas, Nikos Chondros, Panos Diamantopoulos, Christos Patsonakis, and Mema Roussopoulos. On Achieving Interactive Consistency in Real-World Distributed Systems. *Journal of Parallel and Distributed Computing*, 147:220–235, 2021.

[64] Chunyu Mao and Wojciech Golab. Sharding techniques in the era of blockchain. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 343–344, 2021.

[65] Chunyu Mao and Wojciech Golab. Geochain: A locality-based sharding protocol for permissioned blockchains. In *24th International Conference on Distributed Computing and Networking (ICDCN)*, page 70–79, 2023.

[66] Chunyu Mao, Wojciech Golab, and Bernard Wong. Antipaxos: Taking interactive consistency to the next level. In *23rd International Conference on Distributed Computing and Networking (ICDCN)*, pages 128–137, 2022.

[67] Chunyu Mao, Anh-Duong Nguyen, and Wojciech Golab. Performance and Fault Tolerance Trade-offs in Sharded Permissioned Blockchains. In *IEEE International Conference on Blockchain and Cryptocurrency (ICBC) Poster session*, 2020.

[68] Chunyu Mao, Anh-Duong Nguyen, and Wojciech Golab. Performance and fault tolerance trade-offs in sharded permissioned blockchains. In *3rd Conference on Blockchain Research and Applications for Innovative Networks and Services (BRAINS)*, pages 185–192, 2021.

[69] McKinsey&Company. The 2021 McKinsey Global Payments Report. *https://www.mckinsey.com/industries/financial-services/our-insights/the-2021-mckinsey-global-payments-report*, 2021.

[70] Sarah Meiklejohn. Centrally banked cryptocurrencies. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[71] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st edition, 1996.

[72] Esther Mengelkamp, Benedikt Notheisen, Ccarolin Beer, David Dauer, and Christof Weinhardt. A blockchain-based smart grid: towards sustainable local energy markets.

Computer Science—Research and Development. Berlin, Germany: Springer, pages 1–8, August 2017.

[73] Ralph C. Merkle. Protocols for Public Key Cryptosystems. In *Symposium on Security and Privacy*, pages 122–133, 1980.

[74] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. In *ACM Conference on Computer and Communications Security (SIGSAC)*, pages 31–42, 2016.

[75] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

[76] Iulian Moraru, David G. Andersen, and Michael Kaminsky. EPaxos Code Base. *https://github.com/efficient/epaxos*, 2015.

[77] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash system. 2008.

[78] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction. *Princeton University Press, Princeton, NJ, USA*, 2016.

[79] Anh Duong Nguyen. Understanding scalability issues in sharded blockchains. Master's thesis, University of Waterloo, 2020.

[80] Lan N Nguyen, Truc DT Nguyen, Thang N Dinh, and My T Thai. Optchain: Optimal Transactions Placement for Scalable Blockchain Sharding. In *39th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 525–535, 2019.

[81] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (ATC)*, pages 305–319, 2014.

[82] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26(4):631–653, oct 1979.

[83] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[84] Marius Poke and Torsten Hoefler. Dare: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 107–118, 2015.

[85] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. *lightning.network/lightning-network-paper.pdf*, 2016.

[86] Joseph Poon and Thaddeus Dryja. Product Overview. A Technical Overview of xCurrent. *https://ripple.com/files/ripple_product_overview.pdf*, Oct 2017.

[87] Serguei Popov. The tangle. *White paper*, 2018.

[88] Robbert Van Renesse and Deniz Altinbuken. Paxos Made Moderately Complex. *ACM Computing Surveys (CSUR)*, 47(3):1–36, 2015.

[89] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A Scalable and Massively Parallel Consensus Protocol. In *13th ACM International Conference on emerging Networking Experiments and Technologies (CoNEXT)*, pages 426–438, 2017.

[90] Hardik Ruparel, Shreyashree Chiplunkar, Shalin Shah, Madhav Goradia, and Mahesh Shirole. GeoSharding—A Machine Learning-based Sharding Protocol. In *IC-BCT 2019*, pages 105–118. Springer, 2020.

[91] Nuno Santos and André Schiper. Optimizing Paxos with Batching and Pipelining. *Theoretical Computer Science*, 496:170–183, 2013.

[92] Fred B Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[93] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 6 edition, 2011.

[94] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. *Database system concepts*, volume 5. McGraw-Hill New York, 2002.

[95] Nasrin Sohrabi, Zahir Tari, Gathier Voron, Vincent Gramoli, and Qiang Fu. Sazyzz: Scaling azyzzyva to meet blockchain requirements. *IEEE Transactions on Services Computing*, pages 1–14, 2022.

[96] Yee Jiun Song and Robbert van Renesse. Bosco: One-Step Byzantine Asynchronous Consensus. In *22nd International Symposium on Distributed Computing (DISC)*, pages 438–450, 2008.

[97] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, pages 17–33, 2022.

[98] Chrysoula Stathakopoulou, David Tudor, Matej Pavlovic, and Marko Vukolić. Mir-BFT: Scalable and Robust BFT for Decentralized Networks. *Journal of Systems Research*, 2(1), 2022.

[99] Florian Tschorsch and Bjorn Scheuermann. Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. *IEEE Communications Surveys and Tutorials*, 18(3):2084–2123, 2016.

[100] Robbert van Renesse. BoscoChain: Keeping Byzantine Consensus for Blockchains Simple and Flexible. *Presentation at University of Waterloo on Fri, 17 Nov*, 2017.

[101] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Published by Maarten van Steen, 3 edition, 2018.

[102] VISA. Small Business Retail. *https://usa.visa.com/run-your-business/small-business-tools/retail.html*, 2019.

[103] Bitcoin visuals. Bitcoin Transaction sizes. *https://bitcoinvisuals.com/chain-tx-size*, 2022.

[104] Marko Vukolić. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. *Open Problems in Network Security*, pages 112–125, 2016.

[105] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of ACM Symposium on Cloud Computing (SOCC)*, pages 94–107, 2017.

[106] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj a peerto-peer cloud storage network. *White Paper*, December 2014.

[107] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, pages 347–356, 2019.

[108] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling Blockchain via Full Sharding. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 931–948, 2018.